DATA DRIVEN DOCUMENTS

D3.JS

TIPS AND TRICKS

MALCOLM MACLEAN

D3 Tips and Tricks v3.x

Interactive Data Visualization in a Web Browser

Malcolm Maclean

This book is for sale at http://leanpub.com/D3-Tips-and-Tricks

This version was published on 2019-12-24

Leanpub

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

© 2013 - 2019 Malcolm Maclean

Also By Malcolm Maclean

Leaflet Tips and Tricks

Raspberry Pi: Measure, Record, Explore.

Just Enough Linux

Just Enough Co-Authoring in Leanpub

Just Enough ownCloud on a Raspberry Pi

Just Enough Raspberry Pi

Just Enough Ghost on a Raspberry Pi

Just Enough Nagios on a Raspberry Pi

D3 Tips and Tricks v4.x

Never Enough Ice Cream

Raspberry Pi Computing: Temperature Measurement

Simply Leadership

Raspberry Pi Computing: Ultrasonic Distance Measurement

Raspberry Pi Computing: Analog Measurement

PiMetric: Monitoring using a Raspberry Pi

Raspberry Pi Computing: Gas Sensors

Contents

Acknowledgements **1**
Mike 1
Partners, Supporters and Contributors. 1
Proof Reading 1
The d3.js Community 1
Cover art 2
Leanpub 2
Make sure you get the most up to date copy of D3 Tips and Tricks 2

What is d3.js? **3**

Introduction **4**

What do you need to get started? **5**
HTML 5
JavaScript 5
Cascading Style Sheets (CSS) 6
Web Servers 6
PHP 7
Other Useful Stuff 8
Text Editor 8
Getting D3 8
Where to get information on d3.js 10
d3js.org 10
Google Groups 10
Stack Overflow 10
Github 11
bl.ocks.org 11
Twitter 12
Books 12

Starting with a basic graph **13**
HTML 17
CSS 18
D3 JavaScript 20
Setting up the margins and the graph area. 22
Getting the Data 23
Formatting the Date / Time. 26

Setting Scales Domains and Ranges . . . 30
Setting up the Axes . . . 34
Adding data to the line function . . . 38
Adding the SVG Canvas. . . . 38
Actually Drawing Something! . . . 40
Wrap Up . . . 41

Things you can do with the basic graph . . . 43
Adding Axis Labels . . . 43
How to add a title to your graph . . . 51
Smoothing out graph lines . . . 53
Adding grid lines to a graph . . . 61
The grid line CSS . . . 61
Define the grid line functions . . . 63
Draw the lines . . . 64
Make a dashed line . . . 66
Filling an area under the graph . . . 68
CSS for an area fill . . . 68
Define the area function . . . 69
Draw the area . . . 70
Filling an area above the line . . . 71
Adding a drop shadow to allow text to stand out on graphics. . . . 73
CSS for white shadowy background . . . 74
Drawing the white shadowy background. . . . 74
Adding more than one line to a graph . . . 76
Labelling multiple lines on a graph . . . 81
Multiple axes for a graph . . . 83
How to rotate the text labels for the x Axis. . . . 87
Format a date / time axis with specified values . . . 89
Update data dynamically - On Click . . . 91
Adding a Button . . . 91
Updating the data . . . 93
Changes to the d3.js code layout . . . 93
What's happening in the code? . . . 94
Update data dynamically – Automatically . . . 98

Elements, Attributes and Styles . . . 100
The Framework . . . 100
Elements . . . 105
Circle . . . 106
Ellipse . . . 107
Rectangle . . . 108
Line . . . 110
Polyline . . . 111
Polygon . . . 112
Path . . . 113
Clipped Path (AKA clipPath) . . . 114

Text . . . 117
Anchor at the bottom, middle of the text: . . . 117
Anchor at the bottom, right of the text: . . . 119
Anchor at the middle, left of the text: . . . 119
Anchor in the middle, centre of the text: . . . 120
Anchor in the middle, right of the text: . . . 120
Anchor at the top, left of the text: . . . 121
Anchor at the top, middle of the text: . . . 121
Anchor at the top, right of the text: . . . 122
Attributes . . . 123
`x, y` . . . 123
`x1, x2, y1, y2` . . . 124
points . . . 125
`cx, cy` . . . 126
`r` . . . 127
`rx, ry` . . . 128
`transform (translate(x,y), scale(k), rotate(a))` . . . 129
`transform (translate(x,y))` . . . 129
`transform (scale(k))` . . . 130
`transform (rotate(a))` . . . 131
`width, height` . . . 135
`text-anchor` . . . 136
`dx, dy` . . . 137
`textLength` . . . 139
`lengthAdjust` . . . 140
Styles . . . 142
`fill` . . . 143
`stroke` . . . 144
`opacity` . . . 145
`fill-opacity` . . . 146
`stroke-opacity` . . . 147
`stroke-width` . . . 148
`stroke-dasharray` . . . 149
`stroke-linecap` . . . 150
`stroke-linejoin` . . . 152
`writing-mode` . . . 153
`glyph-orientation-vertical` . . . 154
Using styles in Cascading Style Sheets . . . 155

Assorted Tips and Tricks . . . **158**
Change a line chart into a scatter plot . . . 158
Adding tooltips. . . . 161
Transitions . . . 161
Events . . . 162
Get tipping . . . 162
on.mouseover . . . 165

on.mouseout . . . 165
Including an HTML link in a tool tip . . . 165
Moar Links! . . . 167
What are the predefined, named colours? . . . 170
Selecting / filtering a subset of objects . . . 174
Select items with an IF statement. . . . 176
Applying a colour gradient to a line based on value. . . . 178
Applying a colour gradient to an area fill. . . . 182
Show / hide an element by clicking on another element . . . 184
The code . . . 184
Export an image from a d3.js page as a SVG or bitmap . . . 186
Bitmaps . . . 186
Vector Graphics (Specifically SVG) . . . 188
Let's get exporting! . . . 189
Copying the image off the web page . . . 190
Open the SVG Image and Edit . . . 190
Saving as a bitmap . . . 191
Using HTML inputs with d3.js . . . 194
What is an HTML input? . . . 194
Using a `range` input with d3.js . . . 196
The code . . . 196
The explanation . . . 197
Using more than one input . . . 201
The code . . . 201
The explanation . . . 203
Rotate text with an input . . . 205
The explanation . . . 205
Use a `number` input with d3.js . . . 207
Change more than one element with an input . . . 208
The code . . . 208
The explanation . . . 210
Add an HTML table to your graph . . . 212
HTML Tables . . . 213
First the CSS . . . 214
Now the d3.js code . . . 214
A small but cunning change... . . . 217
Explaining the d3.js code (reloaded). . . . 217
Wrap up . . . 219
More table madness: sorting, prettifying and adding columns . . . 220
Add another column of information: . . . 220
Sorting on a column . . . 222
Prettifying (actually just capitalising the header for each column) . . . 222
Add borders . . . 223
Adding web links to d3.js objects . . . 225
It's all about the 'a' and the 'xlink' . . . 226
Adding in the links . . . 226

Making the mouse pointer ignore an object . . . 227
Understanding JavaScript Object Notation (JSON) . . . 229
Using the Yahoo Query Language (YQL) to get data. . . . 233
YQL by example . . . 233
Using Plunker for development and hosting your D3 creations. . . . 236

Bar Charts . . . 241
The data . . . 241
The code . . . 242
The bar chart explained . . . 244

Tree Diagrams . . . 249
What is a Tree Diagram? . . . 249
A simple Tree Diagram explained . . . 251
Styling nodes in a tree diagram . . . 260
Changing the nodes to different shapes . . . 263
Using images as nodes . . . 264
Making a vertical tree diagram . . . 266
Generating a tree diagram from 'flat' data . . . 268
Generating a tree diagram from external data . . . 271
Generating a tree diagram from a CSV file. . . . 273
An interactive tree diagram . . . 274

Force Layout Diagrams . . . 276
What is a Force Layout Diagram? . . . 276
Force directed graph examples. . . . 279
Basic force directed graph showing directionality . . . 281
Directional Force Layout Diagram (Node Highlighting) . . . 292
Directional Force Layout Diagram (varying link opacity) . . . 295
Directional Force Layout Diagram (Unique Node Colour) . . . 297

Bullet Charts . . . 298
Introduction to bullet chart structure . . . 298
D3.js code for bullet charts . . . 299
Adapting and changing bullet chart components . . . 306
Understand your data . . . 306
Add as many individual charts as you want. . . . 307
Add more ranges and measures . . . 307
Updating a bullet chart automatically . . . 309

Mapping with d3.js . . . 312
Examples . . . 312
GeoJSON and TopoJSON . . . 317
Starting with a simple map . . . 318
center . . . 322
scale . . . 323
rotate . . . 324
Zooming and panning a map . . . 326

Displaying points on a map . . . 327
Making maps with d3.js and leaflet.js combined . . . 332
leaflet.js Overview . . . 332
Leaflet map with d3.js objects that scale with the map . . . 333
Leaflet map with d3.js elements that are overlaid on a map . . . 340

D3.js Examples Explained . . . 347
Dynamically retrieve historical stock records via YQL . . . 348
Purpose . . . 348
The code . . . 348
The description . . . 353
Linux Processes via Interactive Tree diagram . . . 357
Purpose . . . 357
The Code . . . 357
Description . . . 363
Multi-line graph with automatic legend and toggling show / hide lines. . . . 367
Purpose . . . 367
The Code . . . 368
Description . . . 370
Nesting the data . . . 370
Applying the colours . . . 372
Adding the legend . . . 373
Making it interactive . . . 376
My Favourite tooltip method for a line graph. . . . 378
Purpose . . . 378
The Code . . . 379
Description . . . 382
Adding the circle to the graph . . . 383
Set the area to capture the mouse movements . . . 384
Determining which date will be highlighted . . . 385
Move the circle to the appropriate position . . . 386
Complex version . . . 387
Code / Explanation . . . 388
Exploring Event Data by Combination Scatter Plot and Interactive Line Graphs . . . 391
Purpose . . . 391
The Code . . . 392
Wrangling the data . . . 392
Sizing Everything Up . . . 394
The Scatter Plot . . . 394
Date and Time Graphs . . . 396
Mouse Movement Information Display . . . 396
Labelling . . . 397
Difference Chart: Science vs Style. . . . 398
Purpose . . . 399
The Code . . . 400
Description . . . 403

Nesting the data . . . 404
Wrangle the data . . . 405
Cheating with the domain . . . 406
`data` vs `datum` . . . 407
Setting up the `clipPaths` . . . 407
Clipping and adding the areas . . . 409
Draw the lines and the axes . . . 411
Adding a bit more to our difference chart. . . . 412
Add a Y axis label . . . 412
Add a title . . . 413
Adding the legend . . . 413
Link the areas . . . 415
The final result . . . 415

Crossfilter, dc.js and d3.js for Data Discovery . . . 417
Introduction to Crossfilter . . . 417
Map-reduce . . . 417
What can crossfilter do? . . . 419
Introduction to dc.js . . . 421
Bar Chart . . . 422
Pie Chart . . . 422
Row Chart . . . 423
Line Chart . . . 423
Bubble Chart . . . 423
Geo Choropleth Chart . . . 424
Data Table . . . 424
Bare bones structure for dc.js and crossfilter page . . . 426
Add a Bar Chart. . . . 435
Position the bar chart . . . 435
Assign the bar chart type . . . 436
Dimension and group the bar chart data . . . 436
Configure the bar chart parameters . . . 437
Just one more thing... . . . 441
Just yet another thing... . . . 441
Position the chart . . . 442
Assign type . . . 442
Dimension and Group . . . 442
Configure chart parameters . . . 442
Add a Line Chart. . . . 444
Position the line chart . . . 444
Assign the line chart type . . . 445
Dimension and group the line chart data . . . 445
Configure the line chart parameters . . . 446
Adding tooltips to a line chart . . . 449
Add a Row Chart. . . . 452
Position the row chart . . . 453

Assign the row chart type . . . 454
Dimension and group the row chart data . . . 454
Configure the row chart parameters . . . 455
Add a Pie Chart. . . . 459
Position the pie chart . . . 460
Assign the pie chart type . . . 461
Dimension and group the pie chart data . . . 461
Configure the pie chart parameters . . . 462
Resetting filters . . . 465
Making the reset label a little bit better behaved. . . . 466
Reset all the charts . . . 468

Using Bootstrap with d3.js . . . 472
What is Bootstrap? . . . 472
Layout grid . . . 473
Interface components . . . 474
Incorporating Bootstrap into your html code. . . . 476
Arranging more than one graph on a web page. . . . 478
First make a page *with* two graphs . . . 478
Arrange the graphs with the same anchor . . . 480
Arrange the graphs with separate anchors . . . 481
How does Bootstrap's grid layout work . . . 483
Arrange more than one d3.js graph with Bootstrap . . . 485
A more complicated Bootstrap layout example . . . 487

Working with GitHub, Gist and bl.ocks.org . . . 495
General stuff about bl.ocks.org . . . 495
Installing the plug-in for bl.ocks.org for easy block viewing . . . 496
Loading a thumbnail into Gist for bl.ocks.org d3 graphs . . . 497
Setting the scene: . . . 497
Enough of the scene setting. Let's git going :-). . . . 499
Wrap up. . . . 501

Appendices . . . 503
Simple Line Graph . . . 503
Graph with Many Features . . . 506
Graph with Area Gradient . . . 511
Bar Chart . . . 514
Linking Objects . . . 517
PHP with MySQL Access . . . 519
Simple Sankey Graph . . . 520
Simple Tree Diagram . . . 523
Interactive Tree Diagram . . . 526
Force Layout Diagram . . . 531
Bullet Chart . . . 536
Map with zoom / pan and cities . . . 539

Acknowledgements

Mike

First and foremost I would like to express my thanks to Mike Bostock, the driving force behind d3.js. His efforts are tireless and his altruism in making his work open and available to the masses is inspiring.

Partners, Supporters and Contributors.

Mike has worked with a crew of like-minded individuals in bringing D3 to the World. Vadim Ogievetsky and Jeffrey Heer share honours for the work on D3: Data-Driven Documents[1] and while there has been a cast of over 40 people contributing to the D3 code base, Jason Davies stands out as the man who has provided a generous portion especially in the area of mapping.

Nick Zhu has created a fantastic resource in dc.js[2] (which is built on top of d3.js and crossfilter) and has been kind enough to provide good advice and permission to include some of his work in the dc.js section.

Advice given by Christophe Viau has been a great help in getting me settled into the on-line world and his energy in managing and directing the D3 community is amazing.

Proof Reading

I am particularly grateful for the assistance given by Filiep Spyckerelle and Robin Bennett who selflessly donated their time and expertise in proofreading above and beyond the call of duty (where this document contains any errors, they are most certainly mine).

The d3.js Community

Big thanks go out to the D3 community. Whether providing advice on Google Groups or Stack Overflow, contributing examples on bl.ocks.org or just giving back in the form of time and effort to similar work. Well done all.

[1] http://vis.stanford.edu/papers/d3
[2] https://github.com/NickQiZhu/dc.js/wiki

Cover art

Out of the blue and in yet another example of the friendly and giving nature of people involved in this community I was contacted by Jose ('Tactician Jenro') who offered to use his skills to design a cover for the book. I think he did a great job and was super helpful. If you think that he could help you out with a project, you can get in touch with him at jenrothetactician@gmail.com or @tacticianjenro.

A sampling of works by Tactician Jenro

Leanpub

Lastly, I want to pay homage to Leanpub[3] who have made the publishing of this document possible. They offer an outstanding service for self-publishing and have made the task of providing and distributing content achievable.

Make sure you get the most up to date copy of D3 Tips and Tricks

If you've received a copy of this book from any location other than Leanpub[4] then it's possible that you haven't got the latest version. Go to https://leanpub.com/D3-Tips-and-Tricks and download the most recent version. After all, it won't cost you anything :-). If you find some value in the work, please consider contributing when you download it so that Leanpub get something for hosting the book (and I'll think of you fondly). Please also be aware that this book is for version 3 of d3.js. If you are considering writing code for version 4 you will want the new edition here[5]. Version 5 of the book is here[6].

[3] https://leanpub.com/
[4] https://leanpub.com/D3-Tips-and-Tricks
[5] https://leanpub.com/d3-t-and-t-v4
[6] https://leanpub.com/d3-t-and-t-v5

What is d3.js?

d3.js[7] (hereafter abridged as D3) is "*a JavaScript library for manipulating documents based on data*".

But that description doesn't do it justice.

D3 is all about helping you to take information and make it more accessible to others via a web browser.

It's a JavaScript library. That means that it's a tool that can be used in conjunction with other tools to get a job done. Those other tools are mainly HTML and CSS (amongst others) but you don't need to know too much about either to use D3 (although it will help :-)).

It's an open framework, which means that there are no hidden mysteries about how it does its magic and it allows others to contribute to a constant cycle of improvement.

It's built to leverage web standards which means that modern browsers don't have to do anything special to use D3, they just have to support the framework that the Internet has adopted for ease of use.

The beauty of D3 is that it allows you to associate data and what appears on the screen in a way that directly links the two. Change the data and you change the object on the screen. D3's trick is to let you set what appears on the screen. A circle, a line, a point on a map, a graph, a bouncing ball, a gradient (and way, way more). Once the data and the object are linked the possibilities are endless.

It won't do everything for you in your quest to create the perfect visualization, but it does give you the ability to achieve that goal.

It bridges the gap between the static display of data and the desire of people to mess about with it. That applies equally to the developer who wants to show something cool and to the end user who wants to be able to explore information interactively.

It was (and still is being) developed by Mike Bostock[8] who has not just spent time writing the code, but writing the documentation[9] for D3 as well. There is an extensive community of supporters who also contribute to the code, provide technical support[10] online[11] and generally have fun creating amazing visualizations[12]. Their contributions are extraordinary (you only have to look at the work of Jason Davies to be amazed).

[7] http://d3js.org/
[8] http://bost.ocks.org/mike/
[9] https://github.com/mbostock/d3/wiki
[10] https://groups.google.com/forum/?fromgroups#!forum/d3-js
[11] http://stackoverflow.com/questions/tagged/d3.js
[12] https://github.com/mbostock/d3/wiki/Gallery

Introduction

I never set out to write treatise on D3…

I am a simple user of this extraordinary framework and when I say simple, I *really* mean I had no idea how to get it to do anything when I started; I needed to do a lot of searching and learned by trial-and-error (emphasis on the errors which were entirely mine). The one thing that I did know was that the example graphics shown by Mike Bostock and others were the sort of graphical goodness that I wanted to play with.

So to get from the point of having no skills whatsoever to the point where I could begin to code up something to display data in a way I wanted, I had to capture the information as I went. The really cool thing about this sort of process is that it doesn't need to occur all at once. You can start with no knowledge whatsoever (or pretty close) and by standing on the shoulders of other's work, you can add building blocks to improve what you're seeing and then change the blocks to adapt and improve.

For example (and this is pretty much how it started). I wanted to draw a line graph, so I imported an example and then got it running locally on my computer. Then I worked out how to change the example data for my data. Then I worked out how to move the Y axis from the right to the left. Then how to make the axis labels larger, change the tick size, make the lines fatter, change the colour, add a label, fill the area under the graph, put the graph in the centre of the page, add a glow to the text to help it stand out, put it in a framework (bootstrap), add buttons to change data sets, animate the transitions between data sets, update the data automatically when it changed, add a pan and zoom feature, turn parts of the graph into hyperlinks to move to other graphs… And then I started on bar graphs :-).

The point to take away from all of this is that any one graph is just a collection of lots of blocks of code, each block designed to carry out a specific function. Pick the blocks you want and implement them.

I found it was much simpler to work on one thing (block) at a time, and this helped greatly to reduce the uncertainty factor when things didn't work as anticipated. I'm not going to pretend that everything I've done while trying to build graphs employs the most elegant or efficient mechanism, but in the end, if it all works on the screen, I walk away happy :-). That's not to say I have deliberately ignored any best practices – I just never knew what they were. Likewise, wherever possible, I have tried to make things as extensible as possible.

You will find that I have typically eschewed a simple "Do this approach" for more of a story telling exercise. This means that some explanations are longer and more flowery than might be to everyone's liking, but there you go, try to be brave :-)

I'm sure most authors try to be as accessible as possible. I'd like to do the same, but be warned… There's a good chance that if you ask me a technical question I may not know the answer. So please be gentle with your emails :-).

Email: d3noobmail+contact@gmail.com

What do you need to get started?

Let's be frank. Not everyone will be inclined to develop a graphic using D3.

However, that doesn't mean that it's beyond those with a little computer savy and a willingness to have a play. Remember failure is your friend (I am fairly sure that I am also related by blood). Just learn from your mistakes and it'll all work out.

So, here in no particular order is a list of good things to know. None of which are essential, but any one (or more) of which will make your life slightly easier.

- HyperText Markup Language (HTML)
- JavaScript
- Cascading Style Sheets (CSS)
- Web Servers
- PHP

DON'T FREAK OUT!

First things first. This isn't rocket science. It's just teh interwebs. We'll take it gently, and I'll be a little more specific in the following sections.

HTML

This stands for HyperText Markup Language and is the stuff that web pages are made of. Check out the definition and other information on Wikipedia[13] for a great overview. Just remember that all you're going to use HTML for is to hold the code that you will use to present your information. This will be as a .html (or .htm) file and they can be pretty simple (we'll look at some in a moment).

JavaScript

JavaScript[14] is what's called a 'scripting language'. It is the code that will be contained inside the HTML file that will make D3 do all its fanciness. In fact, D3 is a JavaScript Library, it's the native language for using D3.

Knowing a little bit about this would be really good, but to be perfectly honest, I didn't know anything about it before I started. I read a book along the way (JavaScript: The Missing Manual[15]

[13] http://en.wikipedia.org/wiki/HTML
[14] http://en.wikipedia.org/wiki/JavaScript
[15] http://shop.oreilly.com/product/9780596515898.do

from O'Reilly) and that helped with context, but the examples that are available for D3 graphics are understandable, and with a bit of trial and error, you can figure out what's going on.

In fact, most of what this collection of information's about is providing examples and explanations for the JavaScript components of D3.

Cascading Style Sheets (CSS)

Cascading Style Sheets[16] (everyone appears to call them 'Style Sheets' or 'CSS') is a language used to describe the formatting (or "look and feel") of a document written in a markup language. The job of CSS is to make the presentation of the components you will draw with D3 simpler by assigning specific styles to specific objects. One of the cool things about CSS is that it is an enormously flexible and efficient method for making everything on the screen look more consistent and when you want to change the format of something you can just change the CSS component and the whole look and feel of your graphics will change.

The wonderful World of Cascading Style Sheets

Full disclosure

I know CSS is a ridiculously powerful tool that would make my life easier, but I use it in a very basic (and probably painful) way. Don't judge me, just accept that the way I've learnt was what I needed to get the job done (this probably means that noobs like myself will find it easier, but where possible try and use examples that include what look like logical CSS structures)

Web Servers

Ok, this can go one of two ways. If you have access to a web server and know where to put the files so that you can access them with your browser, you're on fire. If you're not quite sure, read on…

[16] http://en.wikipedia.org/wiki/Css

A web server will allow you to access your HTML files and will provide the structure that allows it to be displayed on a web browser. There are some simple instructions on the main D3 wiki page[17] for setting up a local server. Or you might have access to a remote one and be able to upload your files. However, for a little more functionality and a whole lot of ease of use, I can thoroughly recommend WampServer as a free and simple way to set up a local web server that includes PHP and a MySQL database (more on those later). Go to the WampServer web page (http://www.wampserver.com/en/) and see if it suits you.

Throughout this document I will be describing the files and how they're laid out in a way that has suited my efforts while using WAMP, but they will work equally well on a remote server. I will explain a little more about how I arrange the files later in the 'Getting D3' section.

WAMP = Windows + Apache + MySQL + PHP

There are other options of course. You could host code on GitHub[18] and present the resulting graphics on bl.ocks.org[19]. This is a great way to make sure that your code is available for peer review and sharing with the wider community.

One such alternative option that I have recently started playing with is Plunker (http://plnkr.co/) This is a lightweight collaborative online editing tool. It's so cool I wrote a special section for it which you can find later in this document. This is definitely worth trying if you want to use something simple without a great deal of overhead. If you like what you see, perhaps consider an alternative that provides a greater degree of capability if you go on to greater d3.js things.

PHP

PHP is a scripting language for the web. That is to say that it is a programming language which is executed when you load web pages and it helps web pages do dynamic things.

You might think that this sounds familiar and that JavaScript does the same thing. But not quite.

JavaScript is designed so that it travels *with* the web page when it is downloaded by a browser (the **client**). However, PHP is executed remotely on the **server** that supplies the web page. This might sound a bit redundant, but it's a big deal. This means that the PHP which is executed doesn't form *part* of the web page, but it can *form* the web page. The implication here is that the web page you are viewing can be altered by the PHP code that runs on a remote server. This is the dynamic aspect of it.

[17] https://github.com/mbostock/d3/wiki
[18] https://github.com/about
[19] http://bl.ocks.org/

In practice, PHP could be analogous to the glue that binds web pages together. Allowing different portions of the web page to respond to directions from the end user.

It is widely recognised not only as a relatively simple language to learn, but also as a fairly powerful one. At the same time it comes into criticism for being somewhat fragmented and sometimes contradictory or confusing. But in spite of any perceived shortcomings, it is a very widely used and implemented language and one for which there is no obvious better option.

Other Useful Stuff

Text Editor

A good text editor for writing up your code will be a real boost. Don't make the fatal mistake of using an office word processor or similar. THEY WILL DOOM YOU TO A LIFE OF MISERY. They add in crazy stuff that you can't even see and never save the files in a way that can be used properly.

Preferably, you should get an editor that will provide some assistance in the form of syntax highlighting which is where the editor knows what language you are writing in (JavaScript for example) and highlights the text in a way that helps you read it. For example, it will change text that might appear as this;

```
// Get the data
d3.tsv("data/data.tsv", function(error, data) {
data.forEach(function(d) {
    d.date = parseDate(d.date);
    d.close = +d.close;
});
```

Into something like this;

```
// Get the data
d3.tsv("data/data.tsv", function(error, data) {
data.forEach(function(d) {
    d.date = parseDate(d.date);
    d.close = +d.close;
});
```

Infinitely easier to use. Trust me.

There are plenty of editors that will do the trick. I have a preference for Geany[20], mainly because it's what I started with and it grew on me :-).

Getting D3

Luckily this is pretty easy.

Go to the D3 repository on github[21] and download the entire repository by clicking on the 'ZIP'

[20] http://www.geany.org/
[21] https://github.com/mbostock/d3

button.

Download the repository as a zip file

What you do with it from here depends on how you're hosting your graphs. If you're working on them on your local PC, then you will want to have the d3.js file in the path that can be seen by the browser. Again, I would recommend WAMP (a local web server) to access your files locally. If you're using WAMP, then you just have to make sure that it knows to use a directory that will contain the d3 directory and you will be away.

The following image is intended to provide a very crude overview of how you can set up the directories.

A potential directory structure for your files

- webserver: Use this as your 'base' directory where you put your files that you create. That way when you open your browser you point to this directory and it allows you to access the files like a normal web site.
- d3: This would be your unzipped d3 directory. It contains all the examples and more importantly the d3.v3.js file that you need to get things going. You will notice in the code examples that follow there is a line like the following;
  `<script type="text/javascript" src="d3/d3.v3.js"></script>`.
  This tells your browser that from the file it is running (one of the graph html files) if it goes into the 'd3' folder it will find the `d3.v3.js` file that it can load.
- data: I use this directory to hold any data files that I would use for processing. For example, you will see the following line in the code examples that follow `d3.tsv("data/data.tsv", function(error, data) {`. Again, that's telling the browser to go into the 'data' directory and to load the 'data.tsv' file.
- js: Often you will find that you will want to include other JavaScript libraries to load. This is a good place to put them.

Where to get information on d3.js

D3 has made huge advances in providing an extensible and practical framework for manipulating data as web objects. At the same time there has been significant increase in information available for people to use it. The following is a far from exhaustive list of sources, but from my own experience it represents a useful subset of knowledge.

d3js.org

d3js.org would be the first port of call for people wanting to know something about d3.js.

From the overview on the main page you can access a dizzying array of examples[22] that have been provided by the founder of d3 (Mike Bostock) and a host of additional developers, artists, coders and anyone who has something to add to the sum knowledge of cool things that can be done with d3.

There is a link to a documentation page[23] that serves as a portal to the ever important API reference, contributed tutorials and other valuable links (some of which I will mention in paragraphs ahead).

The last major link is to the Github repository[24] where you can download d3.js itself.

It is difficult to overstate the volume of available information that can be accessed from d3js.org. It stands alone as the one location that anyone interested in D3 should visit.

Google Groups

There is a Google Group dedicated to discussions on d3.js[25].

In theory this forum is for discussions on topics including visualization design, API design, requesting new features, etc. With a specific direction made in the main header that "*If you want help using D3, please use the d3.js tag on Stack Overflow!*".

In practice however, it would appear that a sizeable proportion of the posts there are technical assistance requests of one type or another. Having said that this means that if you're having a problem, there could already be a solution posted there. However, if at all possible the intention is certainly that people use Stack Overflow, so this should be the first port of call for those types of inquiry.

So, by all means add this group as a favourite and this will provide you with the opportunity to receive emailed summaries of postings or just an opportunity to easily browse recent goings-on.

Stack Overflow

Stack Overflow is a question and answer site whose stated desire is "*to build a library of detailed answers to every question about programming*". Ambitious. So how are they doing? Actually

[22] https://github.com/mbostock/d3/wiki/Gallery
[23] https://github.com/mbostock/d3/wiki
[24] https://github.com/mbostock/d3
[25] https://groups.google.com/forum/?fromgroups#!forum/d3-js

really well. Stack overflow is a fantastic place to get help and information. It's also a great place to help people out if you have some knowledge on a topic.

They have a funny scheme for rewarding users that encourages providing good answers based on readers voting. It's a great example of gamification working well. If you want to know a little more about how it works, check out this page; http://stackoverflow.com/about.

They have a d3.js tag (http://stackoverflow.com/questions/tagged/d3.js) and like Google Groups there is a running list of different topics that are an excellent source of information.

Github

Github[26] is predominantly a code repository and version control site. It is highly regarded for its technical acumen and provides a fantastic service that is broadly used for many purposes. Not the least of which is hosting the code (and the wiki) for d3.js.

Whilst not strictly a site that specialises in providing a Q & A function, there is a significant number of repositories (825 at last count) which mention d3.js. With the help from an astute search phrase, there is potentially a solution to be found there.

The other associated feature of Github is Gist. Gist is a pastebin service (a place where you can copy and past code) that can provide a 'wiki like' feature for individual repositories and web pages that can be edited through a Git repository. Gist plays a role in providing the hub for the bl.ocks.org example hosting service set up by Mike Bostock.

For a new user, Github / Gist can be slightly daunting. It's an area where you almost need to know what's going on to know before you dive in. This is certainly true if you want to make use of its incredible features that are available for hosting code. However, if you want to browse other peoples code it's an easier introduction. Have a look through what's available and if you feel so inclined, I recommend that you learn enough to use their service. It's time well spent.

bl.ocks.org

bl.ocks.org[27] is a viewer for code examples which are hosted on Gist. You are able to load your code into Gist, and then from bl.ocks.org you can view them.

This is a really great way for people to provide examples of their work and there are many who do. However, it's slightly tricky to know what is there. There is a current project[28] being championed by Christophe Viau and others to provide better access to a range of D3 documentation. The early indications are that it will provide a fantastic method of accessing examples and information. Watch that space.

I would describe the process of getting your own code hosted and displaying as something that will be slightly challenging for people who are not familiar with Github / Gist, but again, in terms of visibility of the code and providing an external hosting solution, it is excellent and well worth the time to get to grips with.

[26] https://github.com/
[27] http://bl.ocks.org/
[28] https://groups.google.com/forum/?fromgroups=#!topic/d3-js/g7BxBMUZP8o

Twitter

Twitter provides a great alerting service to inform a large disparate group of people about *stuff*.

It's certainly a great way to keep in touch on an hour by hour basis with people who are involved with d3.js and this can be accomplished in a couple of ways. First, find as many people from the various D3 sites around the web who you consider to be influential in areas you want to follow (different aspects such as development, practical output, educational etc) and follow them. Even better, I found it useful to find a small subset who I considered to be influential people and I noted who they followed. It's a bit '*stalky*' if you're unfamiliar with it, but the end result should be a useful collection of people with something useful to say.

Books

There are only a couple of books that have been released so far on d3.js.

There is "*Getting Started with D3*[29]" by Mike Dewar (O'Reilly Media, June 2012). This will take you through a good set of exercises to develop your D3 skills and is accompanied by downloadable examples.

There is "*Interactive Data Visualization for the Web*[30]" by Scott Murray, (O'Reilly Media, November 2012). Currently this has only been released as an ebook, but is scheduled to be released in print form in 2013. The book is based on his great set of on-line tutorials (http://alignedleft.com/tutorials/).

Of course, there is the original paper that launched D3 "D3: Data-Driven Documents" by Michael Bostock, Vadim Ogievetsky and Jeffrey Heer (IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis), 2011)

[29] http://shop.oreilly.com/product/0636920025429.do
[30] http://ofps.oreilly.com/titles/9781449339739/

Starting with a basic graph

I'll start by providing the full code for a simple graph and then we can go through it piece by piece (The full code for this example is also in the appendices as 'Simple Graph'.).

Here's what the basic graph looks like;

Basic Graph

And here's the code that makes it happen;

```
<!DOCTYPE html>
<meta charset="utf-8">
<style> /* set the CSS */

body { font: 12px Arial;}

path {
    stroke: steelblue;
    stroke-width: 2;
    fill: none;
}

.axis path,
.axis line {
    fill: none;
    stroke: grey;
    stroke-width: 1;
    shape-rendering: crispEdges;
}

</style>
<body>
```

```
<!-- load the d3.js library -->
<script src="http://d3js.org/d3.v3.min.js"></script>

<script>

// Set the dimensions of the canvas / graph
var margin = {top: 30, right: 20, bottom: 30, left: 50},
    width = 600 - margin.left - margin.right,
    height = 270 - margin.top - margin.bottom;

// Parse the date / time
var parseDate = d3.time.format("%d-%b-%y").parse;

// Set the ranges
var x = d3.time.scale().range([0, width]);
var y = d3.scale.linear().range([height, 0]);

// Define the axes
var xAxis = d3.svg.axis().scale(x)
    .orient("bottom").ticks(5);

var yAxis = d3.svg.axis().scale(y)
    .orient("left").ticks(5);

// Define the line
var valueline = d3.svg.line()
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y(d.close); });

// Adds the svg canvas
var svg = d3.select("body")
    .append("svg")
        .attr("width", width + margin.left + margin.right)
        .attr("height", height + margin.top + margin.bottom)
    .append("g")
        .attr("transform",
              "translate(" + margin.left + "," + margin.top + ")");

// Get the data
d3.csv("data/data.csv", function(error, data) {
    data.forEach(function(d) {
        d.date = parseDate(d.date);
        d.close = +d.close;
    });

    // Scale the range of the data
```

```
    x.domain(d3.extent(data, function(d) { return d.date; }));
    y.domain([0, d3.max(data, function(d) { return d.close; })]);

    // Add the valueline path.
    svg.append("path")
        .attr("class", "line")
        .attr("d", valueline(data));

    // Add the X Axis
    svg.append("g")
        .attr("class", "x axis")
        .attr("transform", "translate(0," + height + ")")
        .call(xAxis);

    // Add the Y Axis
    svg.append("g")
        .attr("class", "y axis")
        .call(yAxis);

});

</script>
</body>
```

The full code for this example can be found on github[31], in the appendices of this book or in the code samples bundled with this book (simple-graph.html and data.csv). A live example can be found on bl.ocks.org[32]. Please note that the `<head></head>` tags are omitted which is a common thing for d3 examples (I don't know why). This can cause problems for some browsers in certain conditions.

Once we've finished explaining these parts, we'll start looking at what we need to add in and adjust so that we can incorporate other useful functions that are completely reusable in other diagrams as well.

The end point being something hideous like the following;

[31] https://gist.github.com/d3noob/b3ff6ae1c120eea654b5
[32] http://bl.ocks.org/d3noob/b3ff6ae1c120eea654b5

Graph with lots of 'tricks' incorperated

I say hideous since the graph is not intended to win any beauty prizes, but there are several components to it which some people may find useful (gridlines, area fill, axis label, drop shadow for text, title, text formatting).

So, we can break the file down into component parts. I'm going to play kind of fast and loose here, but never fear, it'll all make sense.

HTML

Here's the HTML portions of the code;

```
<!DOCTYPE html>
<meta charset="utf-8">
<style>

    The CSS is in here

</style>
<body>
<script src="http://d3js.org/d3.v3.min.js"></script>

<script>

    The D3 JavaScript code is here

</script>
</body>
```

Compare it with the full code. It kind of looks like a wrapping for the CSS and JavaScript. You can see that it really doesn't boil down to much at all (that doesn't mean it's not important).

There are plenty of good options for adding additional HTML stuff into this very basic part for the file, but for what we're going to be doing, we really don't need to bother too much.

One thing probably worth mentioning is the line;

```
<script src="http://d3js.org/d3.v3.min.js"></script>
```

That's the line that identifies the file that needs to be loaded to get D3 up and running. In this case the file is sourced from the official d3.js repository on the internet (that way we are using the most up to date version). The D3 file is actually called `d3.v3.min.js` which may come as a bit of a surprise. That tells us that this is version 3 of the d3.js file (the `v3` part) which is an indication that it is separate from the v2 release, which was superseded in late 2012. The other point to note is that this version of d3.js is the minimised version (hence `min`). This means that any extraneous information has been removed from the file to make it quicker to load.

Later when doing things like implementing integration with bootstrap (a pretty layout framework) we will be doing a great deal more, but for now, that's the basics done.

The two parts that we left out are the CSS and the D3 JavaScript.

CSS

The CSS is as follows;

```
body { font: 12px Arial;}

path {
    stroke: steelblue;
    stroke-width: 2;
    fill: none;
}

.axis path,
.axis line {
    fill: none;
    stroke: grey;
    stroke-width: 1;
    shape-rendering: crispEdges;
}
```

Cascading Style Sheets give you control over the look / feel / presentation of web content. The idea is to define a set of properties to objects in the web page.

They are made up of 'rules'. Each rule has a 'selector' and a 'declaration' and each declaration has a property and a value (or a group of properties and values).

For instance in the example code for this web page we have the following rule;

```
body { font: 12px Arial;}
```

`body` is the selector. This tells you that on the web page, this rule will apply to the 'body' of the page. This actually applies to all the portions of the web page that are contained in the 'body' portion of the HTML code (everything between `<body>` and `</body>` in the HTML bit). `{ font: 12px Arial;}` is the declaration portion of the rule. It only has the one declaration which is the bit that is in between the curly braces. So `font: 12px Arial;` is the declaration. The property is `font:` and the value is `12px Arial;`. This tells the web page that the font that appears in the body of the web page will be in 12 px Arial.

Sure enough if we look at the axes of the graph…

x Axis with 12px Arial

We see that the font might *actually* be 12px Arial!

Let's try a test. I will change the Rule to the following;

```
body { font: 16px Arial;}
```

and the result is…

x Axis with 16px Arial

Ahh…. 16px of goodness!

And now we change it to…

```
body { font: 16px times;}
```

and we get…

x Axis with Times font

Hmm… Times font…. I think we can safely say that this has had the desired effect.

So what else is there?

What about the bit that's like;

```
path {
    stroke: steelblue;
    stroke-width: 2;
    fill: none;
}
```

Well, the whole thing is one rule, 'path' is the selector. In this case, 'path' is referring to a line in the D3 drawing nomenclature.

For that selector there are three declarations. They give values for the properties of 'stroke' (in this case colour), 'stroke-width' (the width of the line) and 'fill' (we can fill a path with a block of colour).

So let's change things :-)

```
path {
    stroke: red;
    stroke-width: 5;
    fill: yes;
}
```

Filling of a path

Wow! The line is now red, it looks about 5 pixels wide and it's tried to fill the area (roughly defined by the curve) with a black colour.

It ain't pretty, but it certainly did change. In fact if we go;

```
fill: blue;
```

We'll get...

Filling of a path with added blue!

So the 'fill' property looks pretty flexible. And so does CSS.

D3 JavaScript

The D3 JavaScript part of the code is as follows;

```
var margin = {top: 30, right: 20, bottom: 30, left: 50},
    width = 600 - margin.left - margin.right,
    height = 270 - margin.top - margin.bottom;

var parseDate = d3.time.format("%d-%b-%y").parse;

var x = d3.time.scale().range([0, width]);
var y = d3.scale.linear().range([height, 0]);

var xAxis = d3.svg.axis().scale(x)
    .orient("bottom").ticks(5);

var yAxis = d3.svg.axis().scale(y)
    .orient("left").ticks(5);

var valueline = d3.svg.line()
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y(d.close); });

var svg = d3.select("body")
    .append("svg")
        .attr("width", width + margin.left + margin.right)
        .attr("height", height + margin.top + margin.bottom)
    .append("g")
        .attr("transform",
              "translate(" + margin.left + "," + margin.top + ")");

// Get the data
d3.csv("data.csv", function(error, data) {
    data.forEach(function(d) {
        d.date = parseDate(d.date);
        d.close = +d.close;
    });

    // Scale the range of the data
    x.domain(d3.extent(data, function(d) { return d.date; }));
    y.domain([0, d3.max(data, function(d) { return d.close; })]);

    svg.append("path")                // Add the valueline path.
        .attr("class", "line")
        .attr("d", valueline(data));

    svg.append("g")                       // Add the X Axis
        .attr("class", "x axis")
        .attr("transform", "translate(0," + height + ")")
        .call(xAxis);
```

```
    svg.append("g")                        // Add the Y Axis
        .attr("class", "y axis")
        .call(yAxis);

});
```

Again there's quite a bit of detail in the code, but it's not so long that you can't work out what's doing what.

Let's examine the blocks bit by bit to get a feel for it.

Setting up the margins and the graph area.

The part of the code responsible for defining the canvas (or the area where the graph and associated bits and pieces is placed) is this part.

```
var margin = {top: 30, right: 20, bottom: 30, left: 50},
    width = 600 - margin.left - margin.right,
    height = 270 - margin.top - margin.bottom;
```

This is really (***really***) well explained on Mike Bostock's page on margin conventions here http://bl.ocks.org/3019563[33], but at the risk of confusing you here's my crude take on it.

The first line defines the four margins which surround the block where the graph (as an object) is positioned.

```
var margin = {top: 30, right: 20, bottom: 30, left: 50},
```

So there will be a border of 30 pixels at the top, 20 at the right and 30 and 50 at the bottom and left respectively. Now the cool thing about how these are set up is that they use an object to define everything. That means if you want to do calculations in the JavaScript later, you don't need to put the numbers in, you just use the variable that has been set up. In this case margin.right = 20!

So when we go to the next line;

```
    width = 600 - margin.left - margin.right,
```

the width of the inner block of the canvas where the graph will be drawn is 600 pixels – margin.left – margin.right or 600-50-20 or 530 pixels wide. Of course now you have another variable 'width' that we can use later in the code.

Obviously the same treatment is given to height.

Another cool thing about all of this is that just because you appear to have defined separate areas for the graph and the margins, the whole area in there is available for use. It just makes it really useful to have areas designated for the axis labels and graph labels without having to juggle them and the graph proper at the same time.

So, let's have a play and change some values.

[33] http://bl.ocks.org/3019563

```
var margin = {top: 80, right: 20, bottom: 80, left: 50},
    width = 400 - margin.left - margin.right,
    height = 270 - margin.top – margin.bottom;
```

The effect of changing the margins

Here we've made the graph narrower (400 pixels) but retained the left / right margins and increased the top bottom margins while maintaining the overall height of the canvas. The really cool thing that you can tell from this is that while we shrank the dimensions of the area that we had to draw the graph in, it was still able to dynamically adapt the axes and line to fit properly. That is the really cool part of this whole business. D3 is running in the background looking after the drawing of the objects, while you get to concentrate on how the data looks without too much maths!

Getting the Data

We're going to jump forward a little bit here to the bit of the JavaScript code that loads the data for the graph.

I'm going to go out of the sequence of the code here, because if you know what the data is that you're using, it will make explaining some of the other functions that are coming up much easier.

The section that grabs the data is this bit.

```
d3.csv("data.csv", function(error, data) {
    data.forEach(function(d) {
        d.date = parseDate(d.date);
        d.close = +d.close;
    });
```

In fact it's a combination of a few bits and another piece that isn't shown!, But let's take it one step at a time :-)

There's lots of different ways that we can get data into our web page to turn into graphics. And the method that you'll want to use will probably depend more on the format that the data is in than the mechanism you want to use for importing.

For instance, if it's only a few points of data we could include the information directly in the JavaScript.

That would make it look something like;

```
var data = [
    {date:"1-May-12",close:"58.13"},
    {date:"30-Apr-12",close:"53.98"},
    {date:"27-Apr-12",close:"67.00"},
    {date:"26-Apr-12",close:"89.70"},
    {date:"25-Apr-12",close:"99.00"}
];
```

The format of the data shown above is called JSON (JavaScript Object Notation) and it's a great way to include data since it's easy for humans to read what's in there and it's easy for computers to parse the data out. For a brief overview of JSON there is a separate section in the "Assorted Tips and Tricks Chapter" that may assist.

But if you've got a fair bit of data or if the data you want to include is dynamic and could be changing from one moment to the next, you'll want to load it from an external source. That's when we call on D3's 'Request' functions.

Request Functions

A 'Request' is a function that instructs the browser to reach out and grab some data from somewhere. It could be stored locally (on the web server) or somewhere out in the Internet. There are different types of requests depending on the type of data you want to ingest. Each type of data is formatted with different rules, so the different requests interpret those rules to make sure that the data is returned to the D3 processing in a format that it understands. You could therefore think of the different 'Requests' as translators and the different data formats as being foreign languages.

The different types of data that can be requested by D3 are;

- **text**: A plain old piece of text that has options to be encoded in a particular way (see the D3 API[34]).
- **json**: This is the afore mentioned JavaScript Object Notation.
- **xml**: Extensible Markup Language is a language that is widely used for encoding documents in a human readable forrm.
- **html**: HyperText Markup Language is the language used for displaying web pages.
- **csv**: Comma Separated Values is a widely used format for storing data where plain text information is separated by (wait for it) commas.
- **tsv**: Tab Separated Values is a widely used format for storing data where plain text information is separated by a tab-stop character.

Details on these ingestion methods and the formats for the requests are well explained on the D3 Wiki[35] page. In this particular script we will look at the csv request method.

[34] https://github.com/mbostock/d3/wiki/Requests
[35] https://github.com/mbostock/d3/wiki/Requests

Now, it's important to note that this is not an exclusive list of what can be ingested. If you've got some funky data in a weird format, you can still get it in, but you will most likely need to stand up a small amount of code somewhere else in your page to do the conversion (we will look at this process when describing getting data from a MySQL database).

Back to our request...

```
d3.csv("data.csv", function(error, data) {
    data.forEach(function(d) {
        d.date = parseDate(d.date);
        d.close = +d.close;
    });
```

The first line of that piece of code invokes the d3.csv request (`d3.csv`) and then the function is pointed to the data file that should be loaded (`data.csv`). This is referred to as the 'url' (unique resource locator) of the file. In this case the file is stored locally (in the same directory as the `simple-graph.html` file), but the url could just as easily point to a file somewhere on the Internet.

The format of the data in the data.csv file looks a bit like this;

```
date,close
1-May-12,58.13
30-Apr-12,53.98
27-Apr-12,67.00
26-Apr-12,89.70
25-Apr-12,99.00
```

(although the file is longer (about 26 data points)). The 'date' and the 'close' heading labels are separated by a comma as are each subsequent date and number. Hence the 'comma separated values' :-).

The next part is part of the coolness of JavaScript. With the request made and the file requested, the script is told to carry out a function on the data (which will now be called 'data').

```
function(error, data) {
```

There are actually more things that get acted on as part of the function call, but the one we will consider here is contained in the following lines;

```
    data.forEach(function(d) {
        d.date = parseDate(d.date);
        d.close = +d.close;
    });
```

This block of code simply ensures that all the numeric values that are pulled out of the csv file are set and formatted correctly. The first line sets the data variable that is being dealt with (called slightly confusingly 'data') and tells the block of code that, for each group within the 'data' array it should carry out a function on it. That function is designated 'd'.

```
data.forEach(function(d) {
```

The information in the array can be considered as being stored in rows. Each row consists of two values: one value for 'date' and another value for 'close'.

The function is pulling out values of 'date' and 'close' one row at a time.

Each time it gets a value of 'date' and 'close' it carries out the following operations;

```
d.date = parseDate(d.date);
```

For this specific value of date being looked at (d.date), d3.js changes it into a date format that is processed via a separate function 'parseDate'. (The 'parseDate' function is defined in a separate part of the script, and we will examine that later.) For the moment, be satisfied that it takes the raw date information from the csv file in a specific row and converts it into a format that D3 can then process. That value is then re-saved in the same variable space.

The next line then sets the 'close' variable to a numeric value (if it isn't already) using the '+' operator.

```
d.close = +d.close;
```

This appears to be good practice when the format of the number being pulled out of the data may not mean that it is automagically recognised as a number. This line will ensure that it is.

So, at the end of that section of code, we have gone out and picked up a file with data in it of a particular type (comma separated values) and ensured that it is formatted in a way that the rest of the script can use correctly.

Now, the astute amongst you will have noticed that in the first line of that block of code (`d3.csv("data.csv", function(error, data) {`) we opened a normal bracket (`(`) and a curly bracket (`{`), but we never closed them. That's because they stay open until the very end of the file. That means that all those blocks that occur after the d3.csv bit are referenced to the 'data' array. Or put another way, it uses 'data' to draw stuff!

But anyway, let's get back to figuring what the code is doing by jumping back to the end of the margins block.

Formatting the Date / Time.

One of the glorious things about the World is that we all do things a bit differently. One of those things is how we refer to dates and time[36].

In my neck of the woods, it's customary to write the date as day - month – year. E.g 23-12-2012. But in the United States the more common format would be 12-23-2012. Likewise, the data may be in formats that name the months or weekdays (E.g. January, Tuesday) or combine dates and

[36] http://en.wikipedia.org/wiki/Date_format_by_country

time together (E.g. 2012-12-23 15:45:32). So, if we were to attempt to try to load in some data and to try and get D3 to recognise it as date / time information, we really need to tell it what format the date / time is in.

Does Time Matter?

You might be asking yourself "What's the point?" All you want to do is give it a number and it can sort it out somehow. Well, that is true, but if you want to really bring out the best in your data and to keep maximum flexibility in representing it on the screen, you will want D3 to play to its strengths. And one of those is being able to adjust dynamically with variable time values.

Time for a little demonstration (see what I did there).

We will change our data.csv file so that it only includes two points. The first one and the last one with a separation of a month and a bit.

```
date,close
1-May-12,58.13
26-Mar-12,606.98
```

The graph now looks like this;

Simple line graph

Nothing too surprising here, a very simple graph (note the time scale on the x axis).

Now we will change the later date in the data.csv file so that it is a lot closer to the starting date;

```
date,close
29-Mar-12,58.13
26-Mar-12,606.98
```

So, just a three day difference. Let's see what happens.

Simple line graph over three days

Ahh.... Not only did we not have to make any changes to our JavaScript code, but it was able to recognise the dates were closer and fill in the intervening gaps with appropriate time / day values. Now, one more time for giggles.

This time we'll stretch the interval out by a few years.

```
date,close
29-Mar-21,58.13
26-Mar-12,606.98
```

and the result is...

Simple line graph over several years

Hopefully that's enough encouragement to impress upon you that formatting the time is a *REALLY* good thing to get right. Trust me, it will never fail to impress :-).

Back to formatting.

The line in the JavaScript that parses the time is the following;

```
var parseDate = d3.time.format("%d-%b-%y").parse;
```

This line is used when the `data.forEach(function(d)` portion of the code (that we looked at a couple of pages back) used `d.date = parseDate(d.date)` as a way to take a date in a specific format and to get it recognised by D3. In effect it said "*take this value that is supposedly a date and make it into a value I can work with*".

The function used is the `d3.time.format(specifier)` function where the specifier in this case is the mysterious combination of characters `%d-%b-%y`. The good news is that these are just a combination of directives specific for the type of date we are presenting.

The `%` signs are used as prefixes to each separate format type and the '-' (minus) signs are literals for the actual '-' (minus) signs that appear in the date to be parsed.

The `d` refers to a zero-padded day of the month as a decimal number [01,31].

The `b` refers to an abbreviated month name.

And the `y` refers to the year (without the centuries) as a decimal number.

If we look at a subset of the data from the data.csv file we see that indeed, the dates therein are formatted in this way.

```
1-May-12,58.13
30-Apr-12,53.98
27-Apr-12,67.00
26-Apr-12,89.70
25-Apr-12,99.00
```

That's all well and good, but what if your data isn't formatted exactly like that?

Good news. There are multiple different formatters for different ways of telling time and you get to pick and choose which one you want. Check out the Time Formatting page on the D3 Wiki for a the authoritative list and some great detail, but the following is the list of currently available formatters (from the d3 wiki);

- %a - abbreviated weekday name.
- %A - full weekday name.
- %b - abbreviated month name.
- %B - full month name.
- %c - date and time, as "%a %b %e %H:%M:%S %Y".
- %d - zero-padded day of the month as a decimal number [01,31].
- %e - space-padded day of the month as a decimal number [1,31].
- %H - hour (24-hour clock) as a decimal number [00,23].
- %I - hour (12-hour clock) as a decimal number [01,12].
- %j - day of the year as a decimal number [001,366].
- %m - month as a decimal number [01,12].
- %M - minute as a decimal number [00,59].
- %p - either AM or PM.
- %S - second as a decimal number [00,61].
- %U - week number of the year (Sunday as the first day of the week) as a decimal number [00,53].
- %w - weekday as a decimal number [0(Sunday),6].
- %W - week number of the year (Monday as the first day of the week) as a decimal number [00,53].
- %x - date, as "%m/%d/%y".

- %X - time, as "%H:%M:%S".
- %y - year without century as a decimal number [00,99].
- %Y - year with century as a decimal number.
- %Z - time zone offset, such as "-0700".
- There is also a a literal "%" character that can be presented by using double % signs.

As an example, if you wanted to input date / time formatted as a generic MySQL 'YYYY-MM-DD HH:MM:SS' TIMESTAMP format the D3 parse script would look like;

```
parseDate = d3.time.format("%Y-%m-%d %H:%M:%S").parse;
```

Setting Scales Domains and Ranges

This is another example where, if you set it up right, D3 will look after you forever.

Scales, Ranges and the Ah Ha!" moment.

The "Ah Ha!" moment for me in understanding ranges and scales was after reading Jerome Cukier's great page on 'd3:scales and color[37]'. I thoroughly recommend you read it (and plenty more of the great work by Jerome) as he really does nail the description in my humble opinion. I will put my own description down here, but if it doesn't seem clear, head on over to Jerome's page.

From our basic web page we have now moved to the section that includes the following lines;

```
var x = d3.time.scale().range([0, width]);
var y = d3.scale.linear().range([height, 0]);
```

The purpose of these portions of the script is to ensure that the data we ingest fits onto our graph correctly. Since we have two different types of data (date/time and numeric values) they need to be treated separately (but they do essentially the same job). To examine this whole concept of scales, domains and ranges properly, we will also move slightly out of sequence and (in conjunction with the earlier scale statements) take a look at the lines of script that occur later and set the domain. They are as follows;

```
x.domain(d3.extent(data, function(d) { return d.date; }));
y.domain([0, d3.max(data, function(d) { return d.close; })]);
```

The idea of scaling is to take the values of data that we have and to fit them into the space we have available.

If we have data that goes from 53.98 to 636.23 (as the data we have for 'close' in our csv file does), but we have a graph that is 210 pixels high (height = 270 - margin.top – margin.bottom;) we clearly need to make an adjustment.

[37] http://www.jeromecukier.net/blog/2011/08/11/d3-scales-and-color/

Not only that. Even though our data goes from 53.98 to 636.23, that would look slightly misleading on the graph and it should really go from 0 to a bit over 636.23. It sound's really complicated, but let's simple it up a bit.

First we make sure that any quantity we specify on the x axis fits onto our graph.

```
var x = d3.time.scale().range([0, width]);
```

Here we set our variable that will tell D3 where to draw something on the x axis. By using the d3.time.scale() function we make sure that D3 knows to treat the values as date / time entities (with all their ingrained peculiarities). Then we specify the range that those values will cover (.range) and we specify the range as being from 0 to the width of our graphing area (See! Setting those variables for margins and widths are starting to pay off now!).

Then we do the same for the Y axis.

```
var y = d3.scale.linear().range([height, 0]);
```

There's a different function call (`d3.scale.linear()`) but the .range setting is still there. In the interests of drawing a (semi) pretty picture to try and explain, hopefully this will assist;

Scaling the data to the graph size

I know, I know, it's a little misleading because nowhere have we actually said to D3 this is our data from 53.98 to 636.23. All we've said is when we get the data, we'll be scaling it into this space.

Now hang on, what's going on with the `[height, 0]` part in y axis scale statement? The astute amongst you will note that for the time scale we set the range as `[0, width]` but for this one (`[height, 0]`) the values look backwards.

Well spotted.

This is all to do with how the screen is laid out and referenced. Take a look at the following diagram showing how the coordinates for drawing on your screen work;

Coordinates that the browser expects

The top left hand of the screen is the origin or 0,0 point and as we go left or down the corresponding x and y values increase to the full values defined by height and width.

That's good enough for the time values on the x axis that will start at lower values and increase, but for the values on the y axis we're trying to go against the flow. We want the low values to be at the bottom and the high values to be at the top.

No problem. We just tell D3 via the statement `y = d3.scale.linear().range([height, 0]);` that the larger values (height) are at the low end of the screen (at the top) and the low values are at the bottom (as you most probably will have guessed by this stage, the .range statement uses the format `.range([closer_to_the_origin, further_from_the_origin])`. So when we put the height variable first, that is now associated at the top of the screen.

Coordinates with adjusted ranges

We've scaled our data to the graph size and ensured that the range of values is set appropriately. What's with the domain part that was in this section's title?

Come on, you remember this little piece of script don't you?

```
x.domain(d3.extent(data, function(d) { return d.date; }));
y.domain([0, d3.max(data, function(d) { return d.close; })]);
```

While it exists in a separate part of the file from the scale / range part, it is certainly linked.

That's because there's something missing from what we have been describing so far with the set up of the data ranges for the graphs. We haven't actually told D3 what the range of the data is. That's also the reason this part of the script occurs where it does. It is within the portion where the data.csv file has been loaded as 'data' and it's therefore ready to use it.

So, the `.domain` function is designed to let D3 know what the scope of the data will be. This is what is then passed to the scale function.

Looking at the first part that is setting up the x axis values, it is saying that the domain for the x axis values will be determined by the `d3.extent` function which in turn is acting on a separate function which looks through all the 'date' values that occur in the 'data' array. In this case the `.extent` function returns the minimum and maximum value in the given array.

- `function(d) { return d.date; }` returns all the 'date' values in 'data'. This is then passed to…
- The `.extent` function that finds the maximum and minimum values in the array and then…
- The `.domain` function which returns those maximum and minimum values to D3 as the range for the x axis.

Pretty neat really. At first you might think it was overly complex, but breaking the function down into these components allows additional functionality with differing scales, values and quantities. In short, don't sweat it. It's a good thing.

The x axis values are dates; so the domain for them is basically from the 26th of March 2012 till 1st of May 2012. The y axis is done slightly differently

```
y.domain([0, d3.max(data, function(d) { return d.close; })]);
```

Because the range of values desired on the y axis goes from 0 to the maximum in the data range, that's exactly what we tell D3. The '0' in the `.domain` function is the starting point and the finishing point is found by employing a separate function that sorts through all the 'close' values in the 'data' array and returns the largest one. Therefore the domain is from 0 to 636.23.

Let's try a small experiment. Let's change the y axis domain to use the .extent function (the same way the x axis does) to see what it produces.

The JavaScript for the y domain will be;

```
y.domain(d3.extent(data, function(d) { return d.close; }));
```

You can see apart from a quick copy paste of the internals, all I had to change was the reference to 'close' rather than 'date'.

And the result is…

Graph using .extent for data values

Look at that! The starting point for the y axis looks like it's pretty much on the 53.98 mark and the graph itself certainly touches the x axis where the data would indicate it should.

Now, I'm not really advocating making a graph like this since I think it looks a bit nasty (and a casual observer might be fooled into thinking that the x axis was at 0). However, this would be a useful thing to do if the data was concentrated in a narrow range of values that are quite distant from zero.

For instance, if I change the data.csv file so that the values are represented like the following;

Concentrated data range graph

Then it kind of loses the ability to distinguish between values around the median of the data.

But, if I put in our magic .extent function for the y axis and redraw the graph…

Expanded concentrated data range using .extent

How about that?

The same data as the previous graph, but with one simple piece of the script changed and D3 takes care of the details.

Setting up the Axes

Now we come to our next piece of code;

```
var xAxis = d3.svg.axis().scale(x)
    .orient("bottom").ticks(5);

var yAxis = d3.svg.axis().scale(y)
    .orient("left").ticks(5);
```

I've included both the x and y axes because they carry out the formatting in very similar ways. It's worth noting that this is not the point where the axes get drawn. That occurs later in the piece where the data.csv file has been loaded as 'data'.

D3 has it's own axis component that aims to take the fuss out of setting up and displaying the axes. So it includes a number of configurable options.

Looking first at the x axis;

```
var xAxis = d3.svg.axis().scale(x)
    .orient("bottom").ticks(5);
```

The axis function is called with `d3.svg.axis()`. Then the scale is set using the x values that we set up in the scales, ranges and domains section using `.scale(x)`. Then a curious thing happens, we tell the graph to orientate itself to the bottom of the graph `.orient("bottom")`. If I tell you that "bottom" is the default setting, then you could be forgiven for thinking that technically, we don't need to specify this since it will go there anyway, but it does give us an opportunity to change it to `"top"` to see what happens;

x axis orientated to top

Well, I hope you didn't see that coming, because I didn't. It transpires that what we're talking about is the orientation of the values and ticks about the axis line itself. Ahh… Ok. Useful if your x axis is at the top of your graph, but for this one? Not so useful.

The next part (`.ticks(5)`) sets the number of ticks on the axis. Hopefully you just did a quick count across the bottom of the previous graph and went "Yep, five ticks. Spot on". Well done if you did, but there's a little bit of a sneaky trick up D3's sleeve with the number of ticks on a graph axis.

For instance, here's what the graph looks like when the `.ticks(5)` value is changed to `.ticks(4)`.

Five ticks on the x axis

Eh? Hang on. Isn't that some kind of mistake? There are still five ticks. Yep, sure is! But wait… we can keep dropping the ticks value till we get to two and it will still be the same. At `.ticks(2)` though, we finally see a change.

Two ticks on the x axis

How about that? At first glance that just doesn't seem right, then you have a bit of a think about it and you go "Hmm… When there were 5 ticks, they were separated by a week each, and that stayed that way till we got to a point where it could show a separation of a month.".

D3 is making a command decision for you as to how your ticks should be best displayed. This is great for simple graphs and indeed for the vast majority of graphs. Like all things related to D3, if you really need to do something bespoke, it will let you if you understand enough code.

The following is the list[38] of time intervals that D3 will consider when setting automatic ticks on a time based axis;

- 1-, 5-, 15and 30-second.
- 1-, 5-, 15and 30-minute.
- 1-, 3-, 6and 12-hour.
- 1 and 2-day.
- 1-week.
- 1 and 3-month.
- 1-year.

Just for giggles have a think about what value of ticks you will need to increase to until you get D3 to show more than five ticks.

Hopefully you won't sneak a glance at the following graph before you come up with the right answer.

Ten ticks on the x axis

[38] https://github.com/mbostock/d3/wiki/Time-Scales

Yikes! The answer is 10! And then when it does, the number of ticks is so great that they jumble all over each other. Not looking to good there. However, you could rotate the text (or perhaps slant it) and it could still fit in (that must be the topic of a future how-to). You could also make the graph longer if you wanted, but of course that is probably going to create other layout problems. Try to think about your data and presentation as a single entity.

The code that formats the y axis is pretty similar;

```
var yAxis = d3.svg.axis().scale(y)
    .orient("left").ticks(5);
```

We can change the orientation to `"right"` if we want, but it won't be winning any beauty prizes.

y axis right orientated

Nope. Not a pretty sight.

What about the number of ticks? Well this scale is quite different to the x axis. Formatting the dates using logical separators (weeks, months) was tricky, but with standard numbers, it should be a little easier. In fact, there's a fair chance that you've already had a look at the y axis and seen that there are 6 ticks there when the script is asking for 5 :-)

We can lower the tick number to 4 and we get a logical result.

Three ticks on the y axis

We need to raise the count to 10 before we get more than 6.

Ten ticks on the y axis

Adding data to the line function

We're getting towards the end of our journey through the script now. The next step is to get the information from the array 'data' and to place it in a new array that consists of a set of coordinates that we are going to plot.

```
var valueline = d3.svg.line()
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y(d.close); });
```

I'm aware that the statement above may be somewhat ambiguous. You would be justified in thinking that we already had the data stored and ready to go. But that's not *strictly* correct.

What we have is data in a raw format, we have added pieces of code that will allow the data to be adjusted for scale and range to fit in the area that we want to draw, but we haven't actually taken our raw data and adjusted it for our desired coordinates. That's what the code above does.

The main function that gets used here is the `d3.svg.line()` function[39]. This function uses accessor functions to store the appropriate information in the right area and in the case above they use the x and y accessors (that would be the bits that are `.x` and `.y`). The `d3.svg.line()` function is called a 'path generator' and this is an indication that it can carry out some pretty clever things on its own accord. But in essence its job is to assign a set of coordinates in a form that can be used to draw a line.

Each time this line function is called on, it will go through the data and will assign coordinates to 'date' and 'close' pairs using the 'x' and 'y' functions that we set up earlier (which of course are responsible for scaling and setting the correct range / domain).

Of course, it doesn't get the data all by itself, we still need to actually call the valueline function with 'data' as the source to act on. But never fear, that's coming up soon.

Adding the SVG Canvas.

As the title states, the next piece of script forms and adds the canvas that D3 will then use to draw on.

[39] https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-line

```
var svg = d3.select("body")
    .append("svg")
        .attr("width", width + margin.left + margin.right)
        .attr("height", height + margin.top + margin.bottom)
    .append("g")
        .attr("transform",
              "translate(" + margin.left + "," + margin.top + ")");
```

So what exactly does that all mean?

Well D3 needs to be able to have a space defined for it to draw things. When you define the space it's going to use, you can also give the space you're going to use an identifying name and attributes.

In the example we're using here, we are 'appending' an SVG element (a canvas that we are going to draw things on) to the `<body>` element of the HTML page.

In human talk that means that on the web page and bounded by the `<body>` tag that we saw in the HTML part, we will have an area to draw on. That area will be 'width' plus the left and right margins wide and 'height' plus the top and bottom margins wide.

We also add an element 'g' that is referenced to the top left corner of the actual graph area on the canvas. 'g' is actually a grouping element in the sense that it is normally used for grouping together several related elements. So in this case those grouped elements will have a common reference.

Canvas and margins

(the image above is definitely not to scale, but I hope you get the general idea)

Interesting things to note about the code. The `.attr("stuff in here")` parts are attributes of the appended elements they are part of.

For instance;

```
.append("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
```

tells us that the 'svg' element has a "width" of width + margin.left + margin.right and the "height" of height + margin.top + margin.bottom.

Likewise…

```
.append("g")
    .attr("transform",
          "translate(" + margin.left + "," + margin.top + ")");
```

tells us that the element "g" has been transformed by moving(translating) to the point margin.left, margin.top. Or to the top left of the graph space proper. This way when we tell something to be drawn on our canvas, we can use the reference point "g" to make sure everything is in the right place.

Actually Drawing Something!

Up until now we have spent a lot of time defining, loading and setting up. Good news! We're about to finally draw something!

We jump lightly over some of the code that we have already explained and land on the part that draws the line.

```
svg.append("path")                    // Add the valueline path.
    .attr("d", valueline(data));
```

This area occurs in the part of the code that has the data loaded and ready for action.

The `svg.append("path")` portion adds a new path element . A path element represents a shape that can be manipulated in lots of different ways (see more here: http://www.w3.org/TR/SVG/paths.html[40]). In this case it inherits the 'path' styles from the CSS section and on the following line (`.attr("d", valueline(data));`) we add the attribute "d".

This is an attributer that stands for 'path data' and sure enough the `valueline(data)` portion of the script passes the 'valueline' array (with its x and y coordinates) to the path element. This then creates a svg element which is a path going from one set of 'valueline' coordinates to another.

Then we get to draw in the axes;

```
svg.append("g")                // Add the X Axis
    .attr("class", "x axis")
    .attr("transform", "translate(0," + height + ")")
    .call(xAxis);

svg.append("g")               // Add the Y Axis
    .attr("class", "y axis")
    .call(yAxis);
```

We have covered the formatting of the axis components earlier. So this part is actually just about getting those components drawn onto our canvas.

So both axes start by being appended to the "g" group. Then each has its own classes applied for styling via CSS. If you recall from earlier, they look a little like this;

[40] http://www.w3.org/TR/SVG/paths.html

```
.axis path,
.axis line {
    fill: none;
    stroke: grey;
    stroke-width: 1;
    shape-rendering: crispEdges;
}
```

Feel free to mess about with these to change the appearance of your axes.

On the x axis, we have a transform statement (`.attr("transform", "translate(0," + height + ")")`). If you recall, our point of origin for drawing is in the top left hand corner. Therefore if we want our x axis to be on the bottom of the graph, we need to move (transform) it to the bottom by a set amount. The set amount in this case is the height of the graph proper (height). So, for the point of demonstration we will remove the transform line and see what happens;

x axis transformed to the top of the graph

Yep, pretty much as anticipated.

The last part of the two sections of script (`.call(xAxis);` and `.call(yAxis);`) call the x and y axis functions and initiate the drawing action.

Wrap Up

Well that's it. In theory, you should now be a complete D3 ninja.

OK, perhaps a slight exaggeration. In fact there is a strong possibility that the information I have laid out here is at best borderline useful and at worst laden with evil practices and gross inaccuracies.

But look on the bright side. Irrespective of the nastiness of the way that any of it was accomplished or the inelegance of the code, if the picture drawn on the screen is pretty, you can walk away with a smile. :-)

This section concludes a very basic description of one type of a graphic that can be built with D3. We will look at adding value to it in subsequent chapters.

I've said it before and I'll say it again. This is not a how-to for learning D3. This is how I have managed to muddle through in a bumbling way to try and achieve what I wanted to do. If

some small part of it helps you. All good. Those with a smattering of knowledge of any of the topics I have butchered above (or below) are fully justified in feeling a large degree of righteous indignation. To those I say, please feel free to amend where practical and possible, but please bear in mind this was written from the point of view of someone with no experience in the topic and therefore try to keep any instructions at a level where a new entrant can step in.

Things you can do with the basic graph

The following headings in this section are intended to be a list of relatively simple 'block' type improvements that you can do to your graph to add functionality. The idea is to be able to use the simple graph that was used for the explanation of how D3 worked and just slot in code to add functionality (let's hope it works for you :-)).

I have included the full code for a graph that includes rotated axis label, title, grid lines and filled area as an appendix (Graph with Many Features) for those who would prefer to see the code as a block.

The full code for this example can also be found on github[41] or in the code samples bundled with this book (graph-with-many-features.html and data.csv). A live example can be found on bl.ocks.org[42]

Adding Axis Labels

What's the first thing you get told at school when drawing a graph?

"Always label your axes!"

So, time to add a couple of labels!

First things first (because they're done slightly differently), the x axis. If we begin by describing what we want to achieve, it may make the process of implementing a solution a little more logical.

What we want to do is to add a simple piece of text under the x axis and in the centre of the total span. Wow, that does sound easy.

And it is, but there are different ways of accomplishing it, and I think I should take an opportunity to demonstrate them. Especially since one of those ways is a BAD idea. Lets start with the bad idea first :-).

This is the code we're going to add to the simple line graph script;

[41] https://gist.github.com/d3noob/e1aa61856118edfa624d
[42] http://bl.ocks.org/d3noob/e1aa61856118edfa624d

```
svg.append("text")              // text label for the x axis
    .attr("x", 265 )
    .attr("y",  240 )
    .style("text-anchor", "middle")
    .text("Date");
```

We will put it in between the blocks of script that add the x axis and the y axis.

```
svg.append("g")               // Add the X Axis
    .attr("class", "x axis")
    .attr("transform", "translate(0," + height + ")")
    .call(xAxis);

//        PUT THE NEW CODE HERE!

svg.append("g")              // Add the Y Axis
    .attr("class", "y axis")
    .call(yAxis);
```

Before we describe what's happening, let's take a look at the result;

Date label on x axis

Well, it certainly did what it was asked to do. There's a 'Date' label as advertised! (Yes, I know it's not pretty.) Let's describe the code and then work out why there's a better way to do it.

```
svg.append("text")                // text label for the x axis
    .attr("x", 265 )
    .attr("y", 240 )
    .style("text-anchor", "middle")
    .text("Date");
```

The first line appends a "text" element to our canvas. There is a lot more to learn about "text" elements at the home of the World Wide Web Consortium (W3C)[43]. The next two lines (`.attr("x", 265 )` and `.attr("y", 240 )`) set the attributes for the x and y coordinates to position the text on the canvas.

[43] http://www.w3.org/TR/SVG/text.html#TextElement

The second last line (`.style("text-anchor", "middle")`) ensures that the text 'style' is such that the text is centre aligned and therefore remains nicely centred on the x,y coordinates that we send it to.

The final line (`.text("Date");`) adds the actual text that we are going to place.

That seems really simple and effective and it is. However, the bad part about it is that we have hard coded the location for the date into the code. This means if we change any of the physical aspects of the graph, we will end up having to re-calculate and edit our code. And we don't want to do that.

Here's an example. If I decide that I would prefer to increase the height of the graph by editing the line here;

```
height = 270 - margin.top - margin.bottom;
```

and making the height 350 pixels;

```
height = 350 - margin.top - margin.bottom;
```

The result is as follows;

Hard coded Date label

EVERYTHING about the graph has adjusted itself, except our nasty, hard coded 'Date' label. This is far from ideal and can be easily fixed by using the variables that we set up ever so carefully earlier.

So, instead of;

```
.attr("x", 265 )
.attr("y", 240 )
```

lets let our variables do the walking and use;

```
.attr("x", width / 2 )
.attr("y",  height + margin.bottom)
```

So with this code we tell the script that the 'Date' label will always be halfway across the width of the graph (no matter how wide it is) and at the bottom of the graph with respect to it's height and the bottom margin (remember it uses a coordinates system that increases from the top down).

The end result of using variables is that if I go to an extreme of changing the height and width of my graph to;

```
width = 400 - margin.left - margin.right,
height = 200 - margin.top - margin.bottom;
```

We still finish up with an acceptable result;

Auto adjusting Date label

Well, for the label position at least :-).

So the changes to using variables is just a useful lesson that variables rock and mean that you don't have to worry about your graph staying in relative shape while you change the dimensions. The astute readers amongst you will have learned this lesson very early on in your programming careers, but it's never a bad idea to make sure that users that are unfamiliar with the concept have an indicator of why it's a good idea.

Now the third method that I mentioned at the start of our x axis odyssey. This is not mentioned because it's any better or worse way to implement your script (The reason that I say this is because I'm not sure if it's better or worse.) but because it's sufficiently different to make it look confusing if you didn't think of it in the first place.

So, we'll take our marvellous coordinates code;

```
.attr("x", width / 2 )
.attr("y",  height + margin.bottom)
```

And replace it with a single (longer) line;

```
.attr("transform",
      "translate(" + (width/2) + " ," +
                     (height+margin.bottom) + ")")
```

This uses the `"transform"` attribute to move (translate) the point to place the 'Date' label to exactly the same spot that we've been using for the other two examples (using variables of course).

Why does that line look odd?

The `"translate"` function is done in a 'translate(x,y)' style but it is put on the page in such a way that the verbatim pieces that get passed back are in speech marks and the variables are in the clear (in a manner of speaking). That's why the comma is in speech marks. Additionally, the variables are contained within plus signs. I make the assumption that this is a designator for 'areas where there is variable action going on'. The end result is that if you try to do some maths in that area with a plus sign, it does not appear to work (or at least it didn't for me). That's why I put the variable for `( + (height + margin.bottom) + )` in parenthesis (then I thought I should make the `+ (width / 2) +` part look the same, but actually you can get away without them there).

So, that's the x axis label. Time to do the y axis. The code we're going to use looks like this;

```
svg.append("text")
    .attr("transform", "rotate(-90)")
    .attr("y", 0 - margin.left)
    .attr("x",0 - (height / 2))
    .attr("dy", "1em")
    .style("text-anchor", "middle")
    .text("Value");
```

For the sake of neatness we will put the piece of code in a nice logical spot and this would be following the block of code that added the y axis (but before the closing curly bracket)

```
svg.append("g")                        // Add the Y Axis
    .attr("class", "y axis")
    .call(yAxis);

//        PUT THE NEW CODE HERE!

});
```

And the result looks like this;

y axis label with rotation!

There we go, a label for the y axis that is nicely centred and (gasp!) rotated by 90 degrees! Woah, does the leetness never end! (No. No it does not.)

So, how do we get to this incredible result?

The first thing we do is the same as for the x axis and append a text element to our canvas (`svg.append("text")`).

Then things get interesting.

```
.attr("transform", "rotate(-90)")
```

Because that line rotates everything by -90 degrees. While it's obvious that the text label 'Value' has been rotated by -90 degrees (from the picture), the following lines of code show that we also rotated our reference point (which can be a little confusing).

```
.attr("y", 0 - margin.left)
.attr("x",0 - (height / 2))
```

Let's get graphical to illustrate how this works;

Reference point pre-rotation

Here's our starting position, with x,y in the 0,0 coordinate of the graph drawing area surrounded by the margins.

When we apply a -90 degrees transform we get the equivalent of this;

Reference point after rotation

Here the 0,0 coordinate has been shifted by -90 degrees and the x,y designations are flipped so that we now need to tell the script that we're moving a 'y' coordinate when we would have otherwise been moving 'x'.

Hence, when the script runs…

```
.attr("y", 0 - margin.left)
```

… we can see that this is moving the x position to the left from the new 0 coordinate by the margin.left value.

Likewise when the script runs…

```
.attr("x",0 - (height / 2))
```

… this is actually moving the y position from the new 0 coordinate halfway up the height of the graph area.

I will be the first to admit that this does seem a little confusing. But here's the good part. You really don't need to understand it completely. Simply do what I did when I saw the code. Play with it a bit till you get the result you were looking for. If that means putting in some hard coded numbers and incrementing them to see which way is the new 'up'. Good! Once you work it out, then work out how to get the right variable expression in there and you're set.

In the worst case scenario, simply use the code blocks as shown here and leave well enough alone :-).

Right, we're not quite done yet. The following line has the effect of shifting the text slightly to the right.

```
        .attr("dy", "1em")
```

Firstly the reason we do this is that our previous translation of coordinates means that when we place our text label it sits exactly on the line of `0` – margin.left. But in this case that takes the text to the other side of the line, so it actually sits just outside the boundary of the overall canvas.

The `"dy"` attribute is another coordinate adjustment move, but this time a relative adjustment and the "`1em`" is a unit of measure that equals exactly one unit of the currently specified text point size[44]. So what ends up happening is that the 'Value' label gets shifted to the right by exactly the height of the text, which neatly places it exactly on the edge of the canvas.

The two final lines of this part of the script are the same as for the x axis. They make sure the reference point is aligned to the centre of the text (`.style("text-anchor", "middle")`) and then it prints the text (`.text("Value");`). There, that wasn't too painful.

The astute amongst you (and I'm picking that if you're reading this far into the book, you will definitely qualify as astute or at least suckers for punishment) will notice that the example that I have included in the appendices and online does not have the 'Value' label. Instead it has the text 'Price ($)'and it appears on the area of the graph itself! Well spotted. The technique used above is identical to the one used in the description here, but in the example code we have taken an additional step of demonstrating how to place a white shadowy background under the text (more of that to come in a couple of sections!).

[44] http://en.wikipedia.org/wiki/Em_(typography)

How to add a title to your graph

If you've read through the adding the axis labels section most of this will come as no surprise.

What we want to do to add a title to the graph is to add a text element (just a few words) that will appear above the graph and centred left to right.

The code block we will use will looks like this;

```
svg.append("text")
    .attr("x", (width / 2))
    .attr("y", 0 - (margin.top / 2))
    .attr("text-anchor", "middle")
    .style("font-size", "16px")
    .style("text-decoration", "underline")
    .text("Value vs Date Graph");
```

And the end result will look like this;

Basic graph with title

A nice logical place to put the block of code would be towards the end of the JavaScript. In fact I would put it as the last element we add. So here;

```
svg.append("g")              // Add the Y Axis
    .attr("class", "y axis")
    .call(yAxis);

// PUT THE NEW CODE HERE!

});
```

Now since the vast majority of the code for this block is a regurgitation of the axis labels code, I don't want to revisit that and bloat up this document even more, so I will direct you back to that section if you need to refresh yourself on any particular line. But..... There are a couple of new ones in there which could benefit from a little explanation.

Both of them are style descriptors and as such their job is to apply a very specific style to this element.

```
        .style("font-size", "16px")
        .style("text-decoration", "underline")
```

What they do is pretty self explanatory. Make the text a specific size and underline it. But what is perhaps slightly more interesting is that we have this declaration in the JavaScript code and not in the CSS portion of the file.

Strictly speaking, this is the sort of thing that would be placed in the `<style>` section of the HTML code, but in this case, since it is only going to be used once, we shouldn't feel too bad putting it here.

Smoothing out graph lines

When you draw a line graph, what you're doing is taking two (or more) sets of coordinates and connecting them with a line (or lines). I know that sounds simplistic, but bear with me. When you connect these points, you're telling the viewer of the graph that in between the individual points, you expect the value to vary in keeping with the points that the line passes through. So in a way, you're trying to interpret the change in values that are not shown.

Now this is not strictly true for all graph types, but it does hold for a lot of line graphs.

So… when connecting these known coordinates together, you want to make the best estimate of how the values would be represented. In this respect, sometimes a straight line between points is not the best representation.

For instance. Earlier, when demonstrating the extent function for graphing we showed a graph of the varying values with the y axis showing a narrow range.

Expanded values for a narrow range

The resulting variation of the graph shows a fair amount of extremes and you could be forgiven for thinking that if this represented a smoothly flowing analog system of some kind then some of those sharp peaks and troughs would not be a true representation of how the system or figures varied.

So how should it look? Ahh… The $64,000 question. I don't know :-). You will have a better idea since you are the person who will know your data best. However, what I do know is that D3 has some tricks up its sleeve to help.

We can easily change what we see above into;

Smoothing using "basis"

How about that? And the massive amount of code required to carry out what must be a ridiculously difficult set of calculations?

```
.interpolate("basis")
```

Now, that is slightly unfair because that's the code that YOU need to put in your script, but Mike Bostock probably had to do the mental equivalent of walking across hot coals to get it to work so nicely.

So where does this neat piece of code go? Here;

```
var valueline = d3.svg.line()
    .interpolate("basis")                        // <=== THERE IT IS!
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y(d.close); });
```

So is that it? Nooooo..... There's more! This is one form of interpolation effect that can be applied to your data, but there is a range and depending on your data you can select the one that is appropriate.

Here's the list of available options and for more about them head on over to the D3 wiki[45] and look for 'line.interpolate'.

- linear – Normal line (jagged).
- step-before – a stepping graph alternating between vertical and horizontal segments.
- step-after - a stepping graph alternating between horizontal and vertical segments.
- basis - a B-spline, with control point duplication on the ends (that's the one above).
- basis-open - an open B-spline; may not intersect the start or end.
- basis-closed - a closed B-spline, with the start and the end closed in a loop.
- bundle - equivalent to basis, except a separate tension parameter is used to straighten the spline. This could be really cool with varying tension.
- cardinal - a Cardinal spline, with control point duplication on the ends. It looks slightly more 'jagged' than basis.
- cardinal-open - an open Cardinal spline; may not intersect the start or end, but will intersect other control points. So kind of shorter than 'cardinal'.
- cardinal-closed - a closed Cardinal spline, looped back on itself.
- monotone - cubic interpolation that makes the graph only slightly smoother.

Because in the course of writing this I took an opportunity to play with each of them, I was pleasantly surprised to see some of the effects and it seems like a shame to deprive the reader of the same joy :-). So at the risk of deforesting the planet (so I hope you are reading this in electronic format) here is each of the above interpolation types applied to the same data.

This is also an opportunity to add some reader feedback awesomeness. Many thanks to 'enjalot' for the great suggestion to plot the points of the data as separate circles on the graphs. Since the process of interpolation has the effect of 'interpreting' the trends of the data to the extent that in some cases, the lines don't intersect the actual data much at all.

Each of the following shows the smoothing curve and the data that is used to plot the graph.

[45] https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-line_interpolate

Smoothing using "linear"

Smoothing using "step-before"

Smoothing using "step-after"

Smoothing using "basis"

Smoothing using "basis-open"

Smoothing using "basis-closed"

Smoothing using "bundle"

Smoothing using "cardinal"

Smoothing using "cardinal-open"

Smoothing using "cardinal-closed"

Smoothing using "monotone"

Just in case you're in the mood for another example, here are voronoi tessellations drawn with various d3 line interpolators (the original interactive version by 'shawnbot' can be found here[46]).

[46] http://bl.ocks.org/shawnbot/5970227

First a version using the linear interpolation when each of the points is joined faithfully with a straight line.

Polygon Smoothing using "linear"

Now a version where the polygons are formed with the 'basis-closed' interpolator (note how the lines don't go through the points that describe the bounds of the polygons/blobs).

Polygon Smoothing using "basis-closed"

And lastly, using the 'cardinal-closed' interpolator, while the line travels through each point in the polygon, they overshoot in an effort to maintain a nice curve and the resulting polygon/blobs overlap.

Polygon Smoothing using "cardinal-closed"

So, over to you to decide which format of interpolation is going to suit your data best:-).

Adding grid lines to a graph

Grid lines are an important feature for some graphs as they allow the eye to associate three analogue scales (the x and y axis and the displayed line).

There is currently a tendency to use graphs without grid lines online as it gives the appearance of a 'cleaner' interface, but they are still widely used and a necessary component for graphing.

This is what we're going to draw;

Basic graph with gridlines

Like pretty much everything in this document, the clever parts of this are not my work. I've simply used other peoples cleverness to solve my problems. In this case I think the source of this solution came from the good work of Justin Palmer in his excellent description of the design of a line graph here[47]. However, in retrospect when I've looked back, I'm not sure if I got this right (as I did this quite a while ago when I was less fastidious about noting my sources). In any case, Justin's work is excellent and I heartily recommend it, and here is my implementation of what I think is his work :-)

How to build grid lines?

We're going to use the axis function to generate two more axis elements (one for x and one for y) but for these ones instead of drawing the main lines and the labels, we're just going to draw the tick lines. Really long ticklines (I'm considering calling them long cat[48] lines).

To create them we have to add in 3 separate blocks of code.

1. One in the CSS section to define what style the grid lines will have.
2. One to define the functions that generate the grid lines. And…
3. One to draw the lines.

The grid line CSS

This is the total styling that we need to add for the tick lines;

[47] http://dealloc.me/2011/06/24/d3-is-not-a-graphing-library.html
[48] http://knowyourmeme.com/memes/longcat

```
.grid .tick {
    stroke: lightgrey;
    stroke-opacity: 0.7;
    shape-rendering: crispEdges;
}
.grid path {
          stroke-width: 0;
}
```

Just add this block of code at the end of the current CSS that is in the simple graph template (just before the `</style>` tag).

The CSS here is done in two parts.

The first portion sets the line colour (stroke), the opacity (transparency) of the lines and make sure that the lines are narrow (`crispEdges`).

```
    stroke: lightgrey;
    stroke-opacity: 0.7;
    shape-rendering: crispEdges;
```

The colour is pretty standard, but in using the opacity style we give ourselves the opportunity to use a good shade of colour (if grey actually is a colour) and to juggle the degree to which it stands out a little better.

The second part is the stroke width.

```
    stroke-width: 0;
```

Now it might seem a little weird to be setting the stroke width to zero, but if you don't (and we remove the style) this is what happens;

Axis lines made too thick

If you look closely (compare with the previous picture if necessary) the main lines for the axis have turned thicker. The stroke width style is obviously adding in new (thicker) axis lines and we're not interested in them at the moment. Therefore, if we set the stroke width to zero, we get rid of the problem.

Define the grid line functions

We will need to define two functions to generate the grid lines and they look a little like this;

```
function make_x_axis() {
    return d3.svg.axis()
        .scale(x)
        .orient("bottom")
        .ticks(5)
}

function make_y_axis() {
    return d3.svg.axis()
        .scale(y)
        .orient("left")
        .ticks(5)
}
```

Each function will carry out it's configuration when called from the later part of the script (the drawing part).

A good spot to place the code is just before we load the data with the d3.csv

```
        //   <== Put the functions here!
// Get the data
d3.csv("data.csv", function(error, data) {
    data.forEach(function(d) {
        d.date = parseDate(d.date);
        d.close = +d.close;
    });
}
```

Both functions are almost identical. They give the function a name (make_x_axis and make_y_-axis) which will be used later when the piece of code that draws the lines calls out to them.

Both functions also show which parameters will be fed back to the drawing process when called. Both make sure they use the d3.svg.axis function and then they set individual attributes which make sense.

They make sure they've got the right axis (`.scale(x)` and `.scale(y)`). They set the orientation of the axes to match the incumbent axes (`.orient("bottom")` and `.orient("left")`). And they set the number of ticks to match the number of ticks in the main axis (`.ticks(5)` and `.ticks(5)`). You have the opportunity here to do something slightly different if you want. For instance, think back to when we were setting up the axis for the basic graph and we messed about, seeing how many ticks we could get to appear. If we increase the number of ticks that appear in the grid (lets say to `.ticks(30)` and `.ticks(10)`)) we get the following;

Grid lines with greater divisions

So the grid lines can now show divisions of 50 on the y axis and per day on the x axis :-)

Draw the lines

The final block of code we need is the bit that draws the lines.

```
svg.append("g")
    .attr("class", "grid")
    .attr("transform", "translate(0," + height + ")")
    .call(make_x_axis()
        .tickSize(-height, 0, 0)
        .tickFormat("")
    )

svg.append("g")
    .attr("class", "grid")
    .call(make_y_axis()
        .tickSize(-width, 0, 0)
        .tickFormat("")
    )
```

The first two lines of both the x and y axis grid lines code above should be pretty familiar by now. The first one appends the element to be drawn to the group "g". the second line (`.attr("class", "grid")`) makes sure that the style information set out in the CSS is applied.

The x axis grid lines portion makes a slight deviation from conformity here to adjust its positioning to take into account the coordinates system `.attr("transform", "translate(0," + height + ")")`.

Then both portions call their respective make axis functions (`.call(make_x_axis()` and `.call(make_y_axis())`).

Now comes the really interesting bit.

What you will see if you go to the D3 API wiki[49] is that for the .tickSize function, the following is the format.

[49] https://github.com/mbostock/d3/wiki/SVG-Axes#wiki-tickSize

```
axis.tickSize([major[[, minor], end]])
```

That tells us that you get to specify the size of the ticks on the axes, by the major ticks, the minor ticks and the end ticks (that is to say the lines on the very end of the graph which, in the case of the example we are looking at, aren't there!).

So in our example we are setting our major ticks to a length that corresponds to the full height or width of the graph. Which of course means that they extend across the graph and have the appearance of grid lines! What a neat trick.

Something I haven't done before is to see what would happen if I included the tick lines for the minor and end ticks. So here we go :-)

Disappointment! Where did I go wrong?

Darn! Disappointment. We can see a minor tick line for the y axis, but nothing for the x axis and nothing on the ends. Clearly I will have to run some experiments to see what's going on there (later).

The last thing that is included in the code to draw the grid lines is the instruction to suppress printing any label for the ticks;

```
        .tickFormat("")
```

After all, that would become a bit confusing to have two sets of labels. Even if one was on top of the other. They do tend to become obvious if that occurs (they kind of bulk out a bit like bold text).

And that's it. Grid lines!

Make a dashed line

Dashed lines totally rock!

OK, there may be an element of exaggeration there, but I certainly found it interesting that there didn't seem to be a lot of explanation for a simple bloke like myself to make a dashed line in D3. So for me they rocked :-)

One of the best parts about it is that they're so simple to do!

Literally one line!!!!

So lets imagine that we want to make the line on our simple graph dashed. All we have to do is insert the following line in our JavaScript code here;

```
svg.append("path")
    .attr("class", "line")
    .style("stroke-dasharray", ("3, 3"))  // <== This line here!!
    .attr("d", valueline(data));
```

And our graph ends up like this;

Dashed line for the basic graph

Hey! It's dashtastic!

So how does it work?

Well, obviously `"stroke-dasharray"` is a style for the path element, but the magic is in the numbers.

Essentially they describe the on length and off length of the line. So `"3, 3"` translates to 3 pixels (or whatever they are) on and 3 pixels off. Then it repeats. Simple eh?

So, experiment time :-)

What would the following represent?

"`5, 5, 5, 5, 5, 5, 10, 5, 10, 5, 10, 5`"

Try not to cheat...

Dashed lines for fun

Ahh yes, Mr. Morse would be proud.

And you can put them anywhere. Here's our axes perverted with dashes;

```
svg.append("g")
    .attr("class", "x axis")
    .attr("transform", "translate(0," + height + ")")
    .style("stroke-dasharray", ("3, 3"))
    .call(xAxis);

svg.append("g")
    .attr("class", "y axis")
    .style("stroke-dasharray", ("3, 3"))
    .call(yAxis);
```

When dashed lines go bad

Well... I suppose you can have too much of a good thing. With great power comes great responsibility. Use your dash skills wisely and only for good.

Filling an area under the graph

Lines are all very well and good, but that's not the whole story for graphs. Sometimes you've just got to go with a fill.

Filling an area with a solid colour isn't too hard. I mean we did it by mistake back a few pages when we were trying to draw a line.

But to do it in a nice coherent way is fairly straight forward.

It takes three sections of code in much the same way that we drew our grid lines earlier;

1. One in the CSS section to define what style the area will have.
2. One to define the functions that generate the area. And…
3. One to draw the area.

The end result will looks a bit like this;

Basic graph with an area fill

CSS for an area fill

This is pretty straight forward and only consists of two rules;

```
.area {
    fill: lightsteelblue;
    stroke-width: 0;
}
```

Put them at the bottom of your `<style>` section.

The first one (`fill: lightsteelblue;`) sets the colour of our fill (and in this case we have chosen a lighter shade of the same colour as our line to match it) and the second one (stroke-width: 0;) sets the width of the line that surrounds the area to zero. This last rule is kind of important in making a filled area work well. The whole idea is that the graph is made up of separate elements that will compliment each other. There's the axes, the line and the fill. If we don't tell the code that there is no line surrounding the filled area, it will assume that there is one and add it in like this.

Line surrounding filled area

So what has happened here is that the area element has inherited the line property from the path element and surrounding the area is a 2px wide steelblue line. Not too pretty. Let's not go there.

Define the area function

We need a function that will tell the area what space to fill. This is accessed from the `d3.svg.area` function[50]

The code that we will use is as follows;

```
var area = d3.svg.area()
    .x(function(d) { return x(d.date); })
    .y0(height)
    .y1(function(d) { return y(d.close); });
```

I have placed it in between the axis variable definitions and the line definitions here;

```
var yAxis = d3.svg.axis().scale(y)
    .orient("left").ticks(5);
                        <==== Put the new code here!
var valueline = d3.svg.line()
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y(d.close); });
```

You will notice it looks *INCREDIBLY* similar to the valueline function definition. That's because; while the line definition describes drawing a line that connects a set of coordinates, I imagine the area definition describes drawing two lines that share the same x coordinates, but simultaneously draws two y coordinates, y0 and y1. Then when it's finished drawing the resultant shape, it fills it with the colour of your choosing.

So the only changes to the code are the addition of the `y0` line and the renaming of the `y` line `y1`.

Here's a picture that might help explain;

[50] https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-area

How the area is defined

As should be apparent, the top line (`y1`) follows the valueline line and the bottom line is at the constant 'height' value. Everything in between these lines is what gets filled. The function in this section describes the area.

Draw the area

Now to the money maker.

The final section of code in the area filling odyssey is as follows;

```
svg.append("path")
    .datum(data)
    .attr("class", "area")
    .attr("d", area);
```

We should place this block directly after the domain functions but before the drawing of the valueline path;

```
x.domain(d3.extent(data, function(d) { return d.date; }));
y.domain([0, d3.max(data, function(d) { return d.close; })]);
                            //   <== Area drawing code here!
svg.append("path")
    .attr("class", "line")
    .attr("d", valueline(data));
```

This is actually a pretty good idea to put it there since the various bits and pieces that are drawn in the graph are done so one after the other. This means that the filled area comes first, then the valueline is layered on top and then the axes come last. This is a pretty good sequence since if there are areas where two or more elements overlap, it might cause the graph to look 'wrong'.

For instance, here is the graph drawn with the area added last.

Area overlaps and obscures

You should be able to notice that part of the valueline line has been obscured and the line for the y axis where it coincides with the area is obscured also.

Looking at the code we are adding here, the first line appends a path element (`svg.append("path")`) much like the script that draws the line.

The second line (`.datum(data)`) declares the data we will be utilising for describing the area and the third line (`.attr("class", "area")`) makes sure that the style we apply to it is as defined in the CSS section (under 'area').

The final line (`.attr("d", area);`) declares "d" as the attributer for path data and calls the 'area' function to do the drawing.

And that's it!

Filling an area above the line

Pop Quiz:

How would you go about filling the area *ABOVE* the graph?

Now it might sound a little trite, but believe it or not, this could come in handy. For instance, what if you want to highlight an area that was too high and an area that was too low for a line of data on a graph with an area in the centre where a projected 'normal' set of values should be present?

In this instance, you could fill the lower area as has been demonstrated here, and with a small change you can fill another area with a solid colour above another line.

How is this incredible feat achieved?

Well, remember the code that defined the area?

```
var area = d3.svg.area()
    .x(function(d) { return x(d.date); })
    .y0(height)
    .y1(function(d) { return y(d.close); });
```

All we have to do is tell it that instead of setting the `y0` constant value to the height of the graph (remember, this is the bottom of the graph) we will set it to the constant value that is at the top of the graph. In other words zero (0).

```
.y0(0)
```

That's it.

Fill an area above a line

Now, I'm not going to go over the process of drawing two lines and filling each in different directions to demonstrate the example I described, but this provides a germ of an idea that you might be able to flesh out :-)

Adding a drop shadow to allow text to stand out on graphics.

I've deliberately positioned this particular tip to follow the 'filling an area' description because it provides an opportunity to demonstrate the principle to slightly better effect.

There have been several opportunities where I have wanted to place text overlaid on graphs for convenience sake only to have it look overly messy as the text interferes with the graph.

Is this evil?

Now, I'll be the first to say that the principle of overlaying text on a graph is probably not best practice, but sometimes you've got to do what you've got to do. Besides. Sometimes it's a valid idea. If I remember rightly, the first time I came across this idea, it was being used to highlight text when positioned on bars of a bar graph. So it's not always an evil practice :-).

Anyway, what we'll do is leave the fill in place and place the title back on the graph, but position the title so that it lays on top of the fill like so;

Title lost in the area fill

The additional code for the title is the following and appears just after the drawing of the axes.

```
svg.append("text")
    .attr("x", (width / 2))
    .attr("y", 25 )
    .attr("text-anchor", "middle")
    .style("font-size", "16px")
    .style("text-decoration", "underline")
    .text("Value vs Date Graph");
```

(the only change from the previous title example is the 'y' attribute which has been hard coded to 25 to place it inconveniently on the filled area.)

So, what we want to end up with is something like the following…

A nice white drop shadow effect

In my humble opinion, it's just enough to make the text acceptable :-).

The method that I'll describe to carry this out is designed so that the drop shadow effect can be applied to any text elements in the graph, not the isolated example that we will use here. In order to implement this marvel of utility we will need to make changes in two areas. One in the CSS where we will define a style for white shadowy backgrounds and the second to draw it.

CSS for white shadowy background

The code to add to the CSS section is as follows;

```
text.shadow {
    stroke: white;
    stroke-width: 2.5px;
    opacity: 0.9;
}
```

The first line designates that the style applies to text with a 'shadow' label. The stroke is set to white. the width of the line is set to 2.5px and it is made to be slightly see-through. So by setting the line that surrounds the text to be thick, white and see-through gives it a slightly 'cloudy' effect. If we remove the black text from over the top we get a slightly better look;

A closer look at just the drop shadow

Of course if you want to have a play with any of these settings, you should have a go and see what works best for your graph.

Drawing the white shadowy background.

Now that we've set the style for our background, we need to draw it in.

The code for this should be extremely familiar;

```
svg.append("text")
    .attr("x", (width / 2))
    .attr("y", 25 )
    .attr("text-anchor", "middle")
    .style("font-size", "16px")
    .style("text-decoration", "underline")
    .attr("class", "shadow")                // <=== Here's the different line
    .text("Value vs Date Graph");
```

That's because it's identical to the piece of code that was used to draw the title except for the one line that is indicated above. The reason that it's identical is that what we are doing is placing a white shadow on the graph and then the text on top of it, if it deviated by a significant amount it will just look silly. Of course a slight amount could look effective, in which case adjust the '`x`' or '`y`' attributes.

One of the things I pointed out in the previous paragraph was extremely important. That's the bit that tells you that we needed to place the shadow before we placed the black text. For the same reason that we placed the area fill on first in the area fill example, If black text goes on before the shadow, it will look pretty silly. So place this block of code just before the block that draws the title.

So the line that has been added in is the one that tells D3 that the text that is being drawn will have the white cloudy effect. And at the risk of repeating myself, if you have several text elements that could benefit from this effect, once you have the CSS code in place, all you need to do is duplicate the block that adds the text and add in that single line and voila!

Again the astute amongst you will note that in the example code in the appendices and online, the shadow effect is applied to the Y axis label. Never fear, it's exactly the same principle :-).

Adding more than one line to a graph

All right, we're starting to get serious now. Two lines on a graph is a bit of a step into a different world in one respect. I mean that in the sense that there's more than one way to carry out the task, and I tend to do it one way and not the other mainly because I don't fully understand the other way :-(.

I should stress that that's not because it's more complex, or that it's a bad way, it's just that once I started doing things one way, I haven't come across a need to do things another way. There's a good chance I will have to revisit this decision in the future, but for now I'll keep moving.

So, how are we going to do this? I think that the best way will be to make the executive decision that we have suddenly come across more data and that it is also in our data.csv file (which we'll rename data2.csv just to highlight the difference between the two data sets). In fact it looks a little like this (apologies in advance for the big ugly block of data);

```
date,close,open
1-May-12,68.13,34.12
30-Apr-12,63.98,45.56
27-Apr-12,67.00,67.89
26-Apr-12,89.70,78.54
25-Apr-12,99.00,89.23
24-Apr-12,130.28,99.23
23-Apr-12,166.70,101.34
20-Apr-12,234.98,122.34
19-Apr-12,345.44,134.56
18-Apr-12,443.34,160.45
17-Apr-12,543.70,180.34
16-Apr-12,580.13,210.23
13-Apr-12,605.23,223.45
12-Apr-12,622.77,201.56
11-Apr-12,626.20,212.67
10-Apr-12,628.44,310.45
9-Apr-12,636.23,350.45
5-Apr-12,633.68,410.23
4-Apr-12,624.31,430.56
3-Apr-12,629.32,460.34
2-Apr-12,618.63,510.34
30-Mar-12,599.55,534.23
29-Mar-12,609.86,578.23
28-Mar-12,617.62,590.12
27-Mar-12,614.48,560.34
26-Mar-12,606.98,580.12
```

Three columns, date open and close. The first two are exactly what we have been dealing with all along and the last (open) is our new made up data. Each column is separated by a comma (hence .csv (comma separated values)), which is the format we're currently using to import data.

We should save this as a new file so we don't mess up our previous data, so (as mentioned earlier) let's call it data2.csv.

There is a copy of this file and the sample code at github[51] and in the code samples bundled with this book (dual-line-with-labels.html and data2.csv). A live example can be found on bl.ocks.org[52]. The dual-line-with-labels.html file includes the double lines and also a labelling scheme which we will be describing in a later section.

We will build our new code using our simple graph template to start with, so the immediate consequence of this is that we need to edit the line that was looking for 'data.csv' to reflect the new name.

```
d3.csv("data2.csv", function(error, data) {
```

So when you browse to our new graph's html file, we don't see any changes. It still happily loads the new data, but because it hasn't been told to do anything with it, nothing new happens.

What we need to do now it to essentially duplicate the code blocks that drew the first line for the second line.

The good news is that in the simplest way possible that's just two code blocks. The first sets up the function that defines the new line;

```
var valueline2 = d3.svg.line()
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y(d.open); });
```

You should notice that this block is identical to the block that sets up the function for the first line, except this one is called (imaginatively) valueline2. We should put it directly after the block that sets up the function for valueline.

The second block draws our new line;

```
    svg.append("path")          // Add the valueline2 path.
        .attr("d", valueline2(data));
```

Again, this is identical to the block that draws the first line, except this one is called valueline2. We should put it directly after the block that draws valueline.

After those three small changes, check out your new graph;

[51] https://gist.github.com/d3noob/8603837
[52] http://bl.ocks.org/d3noob/8603837

Two lines, but the same colour

Hey! Two lines! Hmm.... Both being the same colour is a bit confusing. Good news. We can change the colour of the second line by inserting a line that adjusts it's stroke (colour) very simply.

So here's what our new drawing block looks like;

```
svg.append("path")                // Add the valueline2 path.
    .style("stroke", "red")
    .attr("d", valueline2(data));
```

And as if by magic, here's our new graph;

Two lines with two colours

Wow. Right about now, we're thinking ourselves pretty clever. But there's two places where we're not doing things right. We took a simple way, but we took some short cuts that might bite us in the posterior.

The first mistake we made was not ensuring that our variable `"d.open"` is being treated as a number or a string. We're fortunate in this case that it is, but this can't always be assumed. So, this is an easy fix and we just need to put the following (indicated line) in our code;

```
// Get the data
d3.csv("data.csv", function(error, data) {
    data.forEach(function(d) {
        d.date = parseDate(d.date);
        d.close = +d.close;
        d.open = +d.open;        //  <=== Add this line in!
    });
```

The second and potentially more fatal flaw is that nowhere in our code do we make allowance for our second set of data (the second line's values) exceeding our first lines values.

That might not sound too normal straight away, but consider this. What if when we made up our data earlier, some of the new data exceeded our maximum value in our original data? As a means of demonstration, here's what happens when our second line of data has values higher than the first lines;

Two lines but the domain's not right

Ahh.... We're not too clever now.

Good news though, we can fix it!

The problem comes about because when we set the domain for the y axis this is what we put in the code;

```
y.domain([0,d3.max(data, function(d) {return d.close;})]);
```

So that only considers d.close when establishing the domain. With `d.open` exceeding our domain, it just keeps drawing off the graph!

The good news is that 'Bill' has provided a solution for just this problem here[53];

All you need to replace the `y.domain` line with is this;

```
y.domain([0, d3.max(data, function(d) {
    return Math.max(d.close, d.open); })]);
```

[53] http://stackoverflow.com/questions/12732487/d3-js-dataset-array-w-multiple-y-axis-values

It does much the same thing, but this time it returns the maximum of `d.close` and `d.open` (whichever is largest). Good work Bill.

If we put that code into the graph with the higher values for our second line we are now presented with this;

Two lines with everything fitting onto the canvas

And it doesn't matter which of the two sets of data is largest, the graph will always adjust :-)

You will also have noticed that our y axis has auto adjusted again to cope. Clever eh?

Labelling multiple lines on a graph

Our previous example of a graph with multiple lines is a thing of rare beauty, but which line relates to which set of data? We have data that defines values for `open` and `close`, but we don't know which line is which.

In this section we will add labels to our lines so that we know what it what.

This section was inspired by a question from a reader (Arun b.s) of the d3noob.org[54] blog where the question was asked "How can we put text at the end of each line on the graph?".

The question was so good I realised that it had to be part of the book, so here you go :-).

It's actually not too difficult. What we are trying to achieve is to find the position of the end of each line and to add a text label at that position so that the association of proximity denotes the linkage. Of course we're going to go a little further and colour the text so that it's really clear which label belongs with which line, but you get the idea.

Each line requires a single block of script to add the text. The block that adds the `open` label is as follows;

```
svg.append("text")
  .attr("transform", "translate("+(width+3)+","+y(data[0].open)+")")
  .attr("dy", ".35em")
  .attr("text-anchor", "start")
  .style("fill", "red")
  .text("Open");
```

So firstly it appends a textual element to the svg object;

```
svg.append("text")
```

Then it finds the position of the end of the line;

```
  .attr("transform", "translate("+(width+3)+","+y(data[0].open)+")")
```

To do this we use the `transform` and `translate` attribute and find the x position that equates to the end of the graph plus 3 pixels (`(width+3)`) (we add in the three pixels to create a small separation between the end of the line and the label). The `y` position is far more interesting. We need to find the position of the last point in our line for the `open` data. Because the data is in the form of an indexed array and because the data has the latest date at the start of the array, we only need to find the point at the `0` position of the array. This is `data[0].open`. But of course, we also need to adjust our data for our scale and range, so we transform it using the `y` function (in the same way that we do it for the `valueline` and `valueline2` points. So the script to find the point on the screen in the y direction is `y(data[0].open)`.

If our data was arranged with the last date at the end of our data we would have to find the final index point and we would use `y(data[data.length-1].open))`.

Then it's just a matter of aligning and justifying our text correctly;

[54] http://www.d3noob.org/2013/01/adding-more-than-one-line-to-graph-in.html

```
.attr("dy", ".35em")
.attr("text-anchor", "start")
```

Then colouring it the correct colour;

```
.style("fill", "red")
```

And adding out text;

```
.text("Open");
```

We put this block of code after the blocks that add in the axes so that they make sure they're on top of anything else we draw.

The only other small change we want to make is to change the right margin for the graph that we set at the start of our script from 20 to 40 so that there is enough room to add our label without cutting it off.

After that you have a marvellously labelled multi-line graph!

Multi-line graph with labels

The full code for this example can be found on github[55] or in the code samples bundled with this book (dual-line-graph-with-labels.html and data2.csv). A working example can be found on bl.ocks.org[56].

Now, I'd like to pretend that this is perfection, but it isn't. If our lines end too close together, the labels will interfere with each other, so in the ideal world I would include a bit of fanciness to prevent that, but for the purposes of this exercise we can consider ourselves happy.

[55] https://gist.github.com/d3noob/8603837
[56] http://bl.ocks.org/d3noob/8603837

Multiple axes for a graph

Alrighty… Let's imagine that we want to show our wonderful graph with two lines, much like we already have, but that the data that the lines are made from is significantly different in magnitude from the original data (in the example below, the data for the second line has been reduced by approximately a factor of 10 from our original data).

```
date,close,open
1-May-12,58.13,3.41
30-Apr-12,53.98,4.55
27-Apr-12,67.00,6.78
26-Apr-12,89.70,7.85
25-Apr-12,99.00,8.92
24-Apr-12,130.28,9.92
23-Apr-12,166.70,10.13
20-Apr-12,234.98,12.23
19-Apr-12,345.44,13.45
18-Apr-12,443.34,16.04
17-Apr-12,543.70,18.03
16-Apr-12,580.13,21.02
13-Apr-12,605.23,22.34
12-Apr-12,622.77,20.15
11-Apr-12,626.20,21.26
10-Apr-12,628.44,31.04
9-Apr-12,636.23,35.04
5-Apr-12,633.68,41.02
4-Apr-12,624.31,43.05
3-Apr-12,629.32,46.03
2-Apr-12,618.63,51.03
30-Mar-12,599.55,53.42
29-Mar-12,609.86,57.82
28-Mar-12,617.62,59.01
27-Mar-12,614.48,56.03
26-Mar-12,606.98,58.01
```

Now this isn't a problem in itself. D3 will still make a reasonable graph of the data, but because of the difference in range, the detail of the second line will be lost.

One line is dominating the other

What I'm proposing is that we have a second y axis on the right hand side of the graph that relates to the red line.

The mechanism used is based on the great examples put forward by Ben Christensen here[57].

Now… You'll need to concentrate a bit since there are quite a few different bits to change and adapt, but don't despair, they're all quite logical and make sense.

The full code for this example can be found on github[58] or in the code samples bundled with this book (dual-line-dual-axes.html and data2a.csv). A working example can be found on bl.ocks.org[59].

First things first, there won't be space on the right hand side of our graph to show the extra axis, so we should make our right hand margin a little larger.

```
var margin = {top: 30, right: 40, bottom: 30, left: 50},
```

I went for 40 and it seems to fit pretty well.

Then (and here's where the main point of difference for this graph comes in) you want to amend the code to separate out the two scales for the two lines in the graph. This is actually a lot easier than it sounds, since it consists mainly of finding anywhere that mentions `y` and replacing it with `y0` and then adding in a reciprocal piece of code for `y1`.

The idea here is that we will be creating two references for the y axis. One for each column of data. Then when we draw the lines the scales will automatically scale the data correctly (and separately) to our canvas and we will draw two different y axes with the different scales. Believe it or not, it's sounds a lot harder than it is.

Let's get started.

Firstly, change the variable declaration for `y` to `y0` and add in `y1`.

[57] http://benjchristensen.com/2012/05/02/line-graphs-using-d3-js/
[58] https://gist.github.com/d3noob/e34791a32a54e015f57d
[59] http://bl.ocks.org/d3noob/e34791a32a54e015f57d

```
var x = d3.time.scale().range([0, width]);
var y0 = d3.scale.linear().range([height, 0]);
var y1 = d3.scale.linear().range([height, 0]);
```

Then change our `yAxis` declaration to be specific for `y0` and specifically `left`. And add in a declaration for the right hand axis;

```
var yAxisLeft = d3.svg.axis().scale(y0)    //  <==  Add in 'Left' and 'y0'
    .orient("left").ticks(5);

var yAxisRight = d3.svg.axis().scale(y1)   // new declaration for 'Right', 'y1'
    .orient("right").ticks(5);             // and includes orientation .
```

Note the orientation change for the right hand axis.

Now change our valueline declarations so that they refer to the `y0` and `y1` scales.

```
var valueline = d3.svg.line()
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y0(d.close); });    // <== y0
var valueline2 = d3.svg.line()
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y1(d.open); });     // <== y1
```

There are a few different ways for the scaling to work, but we'll stick with the fancy max method we used in the dual line example (although technically it's not required).

```
y0.domain([0, d3.max(data, function(d) { return Math.max(d.close); })]);
y1.domain([0, d3.max(data, function(d) { return Math.max(d.open); })]);
```

Again, here's the `y0` and `y1` changed and added and the maximums for `d.close` and `d.open` are separated out). The final piece of the puzzle is to draw the new axis, but we also want to make a slight change to the original y axis. Since we have two lines and two axes, we need to know which belongs to which, so we can colour code the text in the axes to match the lines;

```
svg.append("g")
    .attr("class", "y axis")
    .style("fill", "steelblue")
    .call(yAxisLeft);

svg.append("g")
    .attr("class", "y axis")
    .attr("transform", "translate(" + width + " ,0)")
    .style("fill", "red")
    .call(yAxisRight);
```

In the above code you can see where we have added in a 'style' change for the yAxisLeft to make it 'steelblue' and a complementary change in the new section for yAxisRight to make that text red.

The yAxisRight section obviously needs to be added in, but the only significant difference is the transform / translate attribute that moves the axis to the right hand side of the graph.

And after all that, here's the result…

Two lines with full range of the domain and two axes

Now, let's not kid ourselves that it's a thing of beauty, but we should console our aesthetic concerns with the warm glow of understanding how the function works :-).

How to rotate the text labels for the x Axis.

The observant reader will recall the problem we had observed earlier when increasing the number of ticks on our x axis to 10. The effect had been to produce a large number of x axis ticks (actually 19) but they had run together and become unreadable.

x axis labels crammed together

We postulated at the time that an answer to the problem might be to rotate the text to provide more space. Well, it's about time we solved that problem.

The answer I found most usable was provided by Aaron Ward on Google Groups[60].

There might be a better way

Now, I'll put a bit of a caveat on this solution to the rotating axis label problem. It looks like it's worked well, but I've only carried out this investigation to the point where I've got something that looks like it's a solution. There may be better or more elegant ways of carrying out the same task, so let Google be your friend if it doesn't appear to be working out for you.

The full code for this example can be found on github[61] or in the code samples bundled with this book (simple-graph-rotated-axis-text.html and data.csv). A working example can be found on bl.ocks.org[62]. The example code also includes the formatting of the x axis ticks in a specific format as described in the next section.

Starting out with our simple graph example, we should increase the number of ticks on the x axis to 10 to highlight the problem in the previous image.

The first substantive change would be a little housekeeping. Because we are going to be rotating the text at the bottom of the graph, we are going to need some extra space to fit in our labels. So we should change our bottom margin appropriately.

```
var margin = {top: 30, right: 40, bottom: 50, left: 50},
```

I found that 50 pixels was sufficient.

The remainder of our changes occur in the block that draws the x axis.

[60] https://groups.google.com/forum/#!msg/d3-js/CRlW0ISbOy4/1sgrE5uS5ysJ
[61] https://gist.github.com/d3noob/ccdcb7673cdb3a796e13
[62] http://bl.ocks.org/d3noob/ccdcb7673cdb3a796e13

```
svg.append("g")
    .attr("class", "x axis")
    .attr("transform", "translate(0," + height + ")")
    .call(xAxis)
    .selectAll("text")
        .style("text-anchor", "end")
        .attr("dx", "-.8em")
        .attr("dy", ".15em")
        .attr("transform", "rotate(-65)");
```

It's pretty standard until the `.call(xAxis)` portion of the code. Here we remove the semicolon that was there so that the block continues with its function.

Then we select all the text elements that comprise the x axis with the `.selectAll("text")`. From this point onwards, we are operating on the text elements associated with the x axis. In effect; the following 4 'actions' are applied to the text labels.

The `.style("text-anchor", "end")` line ensures that the text label has the end of the label 'attached' to the axis tick. This has the effect of making sure that the text rotates about the end of the date. This makes sure that the text all ends up at a uniform distance from the axis ticks.

The `dx` and `dy` attribute lines move the end of the text just far enough away from the axis tick so that they don't crowd it and not too far away so that it appears disassociated. This took a little bit of fiddling to 'look' right and you will notice that I've used the 'em' units to get an adjustment if the size of the font differs.

The final action is kind of the money shot.

The transform attribute applies itself to each text label and rotates each line by -65 degrees. I selected -65 degrees just because it looked OK. There was no deeper reason.

The end result then looks like the following;

Rotated x axis labels

This was a surprisingly difficult problem to find a solution to that I could easily understand (well done Aaron). That makes me think that there are some far deeper mysteries to it that I don't fully appreciate that could trip this solution up. But in lieu of that, enjoy!

Format a date / time axis with specified values

OK then. We've been very clever in rotating our text, but you will notice that D3 has used it's own good judgement as to what format the days / date will be represented as.

Not that there's anything wrong with it, but what if we want to put a specific format of date / time nomenclature as axis labels?

No problem. D3 has your back.

This is actually a pretty easy thing to do, but there are plenty of options for the formatting, so the only really tricky part is deciding what to put where.

But, before we start doing anything we are going to have to expand our bottom margin even more than we did with the rotate the axis labels feature.

```
var margin = {top: 30, right: 40, bottom: 70, left: 50},
```

That should see us right.

Right, now the simple part :-). Changing the format of the label is as simple as inserting the `tickFormat` command into the xAxis declaration a little like this;

```
var xAxis = d3.svg.axis().scale(x)
    .orient("bottom").ticks(10)
    .tickFormat(d3.time.format("%Y-%m-%d")); // insert the tickFormat function
```

An example using this code can be found on github[63] or in the code samples bundled with this book (simple-graph-rotated-axis-text.html and data.csv). A working example can be found on bl.ocks.org[64]. The example code also includes the rotating of the x axis text as described in the previous section.

What the tickFormat allows is the setting of formatting for the tick labels. The `d3.time.format` portion of the code is specifying the exact format of those ticks. This formatting is described using the same arguments that were explained in the earlier section on formatting date time values[65]. That means that the examples we see here (`%Y-%m-%d`) should display the year as a four digit number then a hyphen then the month as a two digit number, then another hyphen, then a two digit number corresponding to the day.

Let's take a look at the result;

[63] https://gist.github.com/d3noob/ccdcb7673cdb3a796e13
[64] http://bl.ocks.org/d3noob/ccdcb7673cdb3a796e13
[65] https://github.com/mbostock/d3/wiki/Time-Formatting

Format change for the x axis labels

There we go! You should be able to see this file in the downloads section on d3noob.org with the general examples as formatted-date-time-axis-labels.html.

So how about we try something a little out of the ordinary (extreme)?

How about the full weekday name (`%A`), the day (`%d`), the full month name (`%B`) and the year (`%Y`) as a four digit number?

```
.tickFormat(d3.time.format("%A %d %B %Y"));
```

We will also need some extra space for the bottom margin, so how about 140?

```
var margin = {top: 30, right: 40, bottom: 140, left: 50},
```

and....

Extreme format change for the x axis labels

Oh yeah... When axis ticks go bad...

But seriously, that does work as a pretty good example of the flexibility available.

Update data dynamically - On Click

OK, you're going to enjoy this section. Mainly because it takes the traditional graph that we know, love and have been familiar with since childhood and adds in an aspect that that has been missing for most of your life.

Animation!

Graphs are cool. Seeing information represented in a graphical way allows leaps of understanding that are difficult or impossible to achieve from raw data. But in this crazy ever-changing world, a static image is only as effective as the last update. The ability to being able to have the most recent data represented in your graph and to have it occur automatically provides a new dimension to traditional visualizations.

Interestingly enough, part of the reason for moving from D3's predecessor Protovis[66] was the ability to provide greater control and scope to animating data.

So what are we going to do?

First we'll spend a bit of time setting the scene. We'll add a button to our basic graph file so that we can control when our animation occurs, we'll generate a new data set so that we can see how the data changes easily, then we'll shuffle the code about a bit to make it do its magic. While we're shuffling the code we'll take a little bit of time to explain what's going on with various parts of it that are different to what we might have seen thus far. Then we'll change the graph to update automatically (on a schedule) when the data changes.

One of the problems with writing a manual about a moving object is that it's difficult to represent that movement on a written page, so where there is something animated occurring, I will provide all the code that I'm using so that you can try it at home and have an online version as well.

The code for this example can be found on github[67] or in the code samples bundled with this book (data-load-button.html, data.csv and data-alt.csv). A working example can be found on bl.ocks.org[68].

Adding a Button

It's all well and good animating your data, but if you don't know when it's supposed to happen or what should happen, it's a little difficult to evaluate how successful you've been.

To make life easy, we're going to take some of the mystery out of the equation (don't worry, we'll put it back later) and add a button to our graph that will give you control over when your graph should update it's data. When complete it should look like this;

[66] http://mbostock.github.com/d3/tutorial/protovis.html
[67] https://gist.github.com/d3noob/7030f35b72de721622b8
[68] http://bl.ocks.org/d3noob/7030f35b72de721622b8

A graph with a button!

To add a button, we will take our simple-graph.html example and just after the `<body>` tag we add the following code;

```
<div id="option">
    <input name="updateButton"
           type="button"
           value="Update"
           onclick="updateData()"
    />
</div>
```

The HTML `<div>` element (or HTML Document Division Element) is used to assign a division or section in an HTML document. We use it here as it's good practice to keep sections of your HTML document distinct so that it's easier to perform operations them at a later date.

In this case we have given the div the identifier "option" so that we can refer to it later if we need to (embarrassingly, we won't be referring to it at all, but it's good practice none the less).

The following line adds our button using the HTML `<input>` tag. The `<input>` tag has a wide range of attributes (options) for allowing user input. Check out the links to w3schools[69] and Mozilla[70] for a whole lot of reading.

In our `<input>` line we have four different attributes;

- name
- type
- value
- onclick

Each of these attributes modifies the `<input>` function in some way so that our button does what we want it to do.

[69] http://www.w3schools.com/tags/tag_input.asp
[70] https://developer.mozilla.org/en-US/docs/HTML/Element/Input

name:
This is the name of the control (in this case a button) so that we can reference it in other parts of our HTML script.

type:
Probably the most important attribute for a button, this declares that our type of input will be a button! There are *heaps* of other options for `type` which would form a significant section in itself.

value:
For a `button` input type, this is the starting value for our button and forms the label that our button will have.

onclick:
This is not an attribute that is specific to the `<input>` function, but it allows the browser to capture a mouse clicking event when it occurs and in our case we tell it to run the `updateData()` function (which we'll be seeing more of soon).

Updating the data

To make our first start at demonstrating changing the data, we'll add another data file to our collection. We'll name it `data-alt.csv` (you can find it in the sample code collection that can be downloaded with the book on Leanpub). This file changes our normal data (only the values, not the structure) just enough to see a movement of the time period of the graph and the range of values on the y axis (this will become really obvious in the transition).

Temporary measure only

We'll only use this file while we want to demonstrate that dynamic updating really does work. Ultimately we will just use the one file and rely on an external process updating that file to provide the changing data.

Changes to the d3.js code layout

While going through the process of working out how to do this, the iterations of my code were mostly horrifying to behold. However, I think my understanding has improved sufficiently to allow only a slight amendment to our simple-graph.html JavaScript code to get this going.

What we should do is add the following block of code to our script towards the end of the file just before the `</script>` tag;

```
function updateData() {

    // Get the data again
    d3.csv("data-alt.csv", function(error, data) {
                data.forEach(function(d) {
                     d.date = parseDate(d.date);
                     d.close = +d.close;
            });

            // Scale the range of the data again
            x.domain(d3.extent(data, function(d) { return d.date; }));
            y.domain([0, d3.max(data, function(d) { return d.close; })]);

    // Select the section we want to apply our changes to
    var svg = d3.select("body").transition();

    // Make the changes
        svg.select(".line")   // change the line
            .duration(750)
            .attr("d", valueline(data));
        svg.select(".x.axis") // change the x axis
            .duration(750)
            .call(xAxis);
        svg.select(".y.axis") // change the y axis
            .duration(750)
            .call(yAxis);

    });
}
```

What's happening in the code?

There are several new concepts and techniques in this block of code for us to go through but we'll start with the overall wrapper for the block which is a function call.

The entirety of our JavaScript code that we're adding is a function called `updateData`. This is the subject of the first line in the code above (and the last closing curly bracket). It is called from the only other piece of code we've added to the file which is the button in the HTML section. So when that button is clicked, the `updateData` function is carried out.

Repeatability

It's worth noting that while our `updateData` function only appears to work the once when you first click the button, in fact every time the button is pushed the `updateData` function is carried out. It's just that since the data doesn't change after the first click, you never see any change.

Then we get our new data with the block that starts with `d3.csv("data-alt.csv"`. This is a replica of the block in the main part of the code with one glaring exception. It is getting the data from our new file called `data-alt.csv`. However, one thing it's doing that bears explanation is that it's loading data into an array that we've already used to generate our line. At a point not too far from here (probably the next page) we're going to replace the data that made up our line on the page with the new data that's just been loaded.

We then set the scale and the range again using the `x.domain` and `y.domain` lines. We do this because it's more than possible that our data has exceeded or shrunk with respect to our original domains so we recalculate them using our new data. The consequence of not doing this would be a graph that could exceed it's available space or be cramped up.

Then we assign the variable svg to be our selection of the `"body"` div (which means the following actions will only be carried out on objects within the `"body"` div.

Selection Study.

Selections are a very important topic and if reading Google Groups and Stack Overflow are anything to go by they are also a much misunderstood feature of D3. I won't claim to be in any better position to describe them, but I would direct readers to a description of nested selections by Mike Bostock (http://bost.ocks.org/mike/nest/) and a video tutorial by Ian Johnson (http://blog.visual.ly/using-selections-in-d3-to-make-data-driven-visualizations/).

The other part of that line is the transition command (`.transition()`). This command goes to the heart of animating dynamic data and visualizations and is a real treasure.

Transition Training

I will just be brushing the surface of the subject of transitions in d3.js, and I will certainly not do the topic the justice it deserves for in depth animations. I heartily recommend that you take an opportunity to read Mike Bostock's "Path Transitions" (http://bost.ocks.org/mike/path/), bar chart tutorial (http://mbostock.github.com/d3/tutorial/bar-2.html) and Jerome Cukier's "Creating Animations and Transitions with D3" (http://blog.visual.ly/creating-animations-and-transitions-with-d3-js/). Of course, one of the main resources for information on transitions is also the D3 wiki (https://github.com/mbostock/d3/wiki/Transitions).

As the name suggests, a transition is a method for moving from one state to another. In its simplest form for a d3.js visualisation, it could mean moving an object from one place to another, or changing an object's properties such as opacity or colour. In our case, we will take our data which is in the form of a line, and change some of that data. And when we change the data we will get d3 to manage the change via a transition. At the same time (because we're immensely clever) we will also make sure we change the axes if they need it.

So in short, we're going to change this…

The initial set of data

... into this...

'Updated' data

Obviously the line values have changed, and both axes have changed as well. And using a properly managed transition, it will all occur in a smooth ballet :-).

So, looking at the short block that manages the line transition;

```
svg.select(".line")   // change the line
    .duration(750)
    .attr("d", valueline(data));
```

We select the `".line"` object and since we've already told the script that `svg` is all about the transition (`var svg = d3.select("body").transition();`) the attributes that follow specify how the transition for the line will proceed. In this case, the code describes the length of time that the transition will take as 750 milliseconds (`.duration(750)`) and uses the new data as transcribed by the `valueline` variable from the original part of the script (`.attr("d", valueline(data));`).

The same is true for both of the subsequent portions of the code that change the x and y axes. We've set both to a transition time of 750 milliseconds, although feel free to change those values (make each one different for an interesting effect).

Other attributes for the transition that we could have introduced would be a delay (`.delay(500)`, perhaps to stagger the movements) and more interestingly an easing attribute (`.ease(type[, arguments…])`) which will have the effect of changing how the movement of a transition appears (kind of like a fast-slow-fast vs linear, but with lots of variations).

But for us we'll survive with the defaults.

In theory, you've added in your new data file (`data-alt.csv`) and made the two changes to the simple graph file (the HTML block for the button and the JavaScript one for the `updateData` function). The result has been a new beginning in your wonderful d3 journey!

Revert the data

If you fancy a quick test, consider what you would need to do to add another button that was labelled 'Revert' which, when pressed changed the graph back to the original data (so that you could merrily press 'Update' and 'Revert' all day if you wanted).
I have loaded a simplistic version of the graph that will do this as an example can be found on github[71] or in the code samples bundled with this book (data-load-revert-button.html, data.csv and data-alt.csv). A working example can be found on bl.ocks.org[72].. There are more elegant ways to code this, but the example I give is pretty easy to follow.

[71] https://gist.github.com/d3noob/a048edddbf83bff03a34
[72] http://bl.ocks.org/d3noob/a048edddbf83bff03a34

Update data dynamically – Automatically

I have no doubt that the excitement of updating your data and graph with the magic of buttons is quite a thrill. But believe it or not, there's more to come.

In the example we're going to demonstrate now, there are no buttons to click, the graph will simply update itself when the data changes.

I know, I know. It's like magic!

So the sort of usage scenario that you would be putting this to is when you had a dashboard type display or a separate window just for the purposes of showing a changing value like a stock ticker or number of widgets sold (where the number was changing frequently).

So, how to create the magic?

Starting with the data-load-button.html file, firstly we should remove the button, so go ahead and delete the button block that we had in the HTML section (the bit that looked like this...).

```
<div id="option">
    <input name="updateButton"
                 type="button"
                value="Update"
                onclick="updateData()" />
</div>
```

Now, we have two references in our JavaScript where we load our data. One loads `data.csv` initially, then when the button was pushed, we loaded `data-alt.csv`. We're going to retain that arrangement for the moment, because we want to make sure we can see something happening, but ultimately, we would just have them referencing a single file.

So, the magic piece of script that will do your updating is as follows;

```
var inter = setInterval(function() {
                updateData();
        }, 5000);
```

And we should put that just above the `function updateData() {` line in our code.

The key to this piece of code is the `setInterval` function which will execute specified code (in this case it's `updateData();` which will go and read in our new information) over and over again at a set interval (in this case 5000 milliseconds (`}, 5000);`)).

I honestly wish it was harder, but sadly it's that simple. You now have in your possession the ability to make your visualizations do *stuff* on a regular basis, all by themselves!

There is a copy of this sample code and the data files at github[73] and in the code samples bundled with this book (data-load-automatic.html, data.csv and data-alt.csv). A live example can be found on bl.ocks.org[74].

[73] https://gist.github.com/d3noob/6bd13f974d6516f3e491
[74] http://bl.ocks.org/d3noob/6bd13f974d6516f3e491

How to test?

Well, just load up your new file in a browsers. After an interval of 5 seconds, you should see the graph change all by itself. How cool is that?

You know it gets better though…

If you open your data-alt.csv file and change a value (increase one of the `close` values by a factor of 10 or something equally noticeable). Then save the file. Keep an eye on your graph. Before 5 seconds is up it should have changed to reflect your new data.

There is a possibility that your browser may have decided to cache the data from the data-alt.csv file, in which case you can tell it to stop that nonsense by going into the settings and clearing the cache.

Elements, Attributes and Styles

This chapter is intended to provide an overview of some of the simpler things that d3.js can do, but in a way that may help some understand a little more about how images can be added to a web page and how they can be manipulated.

Loosely speaking we will look at how objects (elements (like circles, rectangles, lines and even text) can be declared and added to a page, how their attributes in relation to the page (position, size, shape, actions) can be changed and how their style (colour, width, transparency) can be applied.

As we go through the explanation of different changes that can be applied to different elements there will be a small amount of repetition where there is cross-over with related drawing features. Please be patient :-). The aim is to have each section as complete in its own right as practical.

The Framework

To be able to demonstrate how these three related aspects of drawing objects work we will have to use a small, simple script to draw them in your web browser.

We will just take a moment to explain the script that draws a circle.

Here's the contents of the file in it's entirety. I have imaginatively called it `circle.html`.

```
<!DOCTYPE html>
<meta charset="utf-8">

<body>

<!-- load the d3.js library -->
<script type="text/javascript" src="d3/d3.v3.js"></script>

<script>

var holder = d3.select("body") // select the 'body' element
      .append("svg")           // append an SVG element to the body
      .attr("width", 449)      // make the SVG element 449 pixels wide
      .attr("height", 249);    // make the SVG element 249 pixels high

// draw a circle
holder.append("circle")        // attach a circle
    .attr("cx", 200)           // position the x-center
    .attr("cy", 100)           // position the y-center
    .attr("r", 50);            // set the radius
```

```
</script>

</body>
```

Please feel free to jump ahead slightly if you understand how a HTML file with JavaScript goes together :-).

The HTML part of the file can be thought of as a wrapper for the JavaScript that will draw our circle. These are the HTML parts here…

```
<!DOCTYPE html>
<meta charset="utf-8">

<body>

<!-- load the d3.js library -->
<script type="text/javascript" src="d3/d3.v3.js"></script>

<script>

</script>

</body>
```

This portion of the file is built using HTML 'tags'. These will set up the environment for the Javascript.

The tags tell the web browser what sort of language is being used and the type of characters used to write the code…

```
<!DOCTYPE html>
<meta charset="utf-8">
```

Areas of the code are labelled.

Like the body…

```
<body>

In this area we can put the stuff that will be
displayed on our web page.

</body>
```

And the place where we put the JavaScript…

```
<script>

Our d3.js code will go here.

</script>
```

We even load an external file that contains JavaScript that will help run our code.

```
<!-- load the d3.js library -->
<script type="text/javascript" src="d3/d3.v3.js"></script>
```

Yes, that's the line that loads d3.js. Once it's loaded we can use the instructions that it makes available to make other JavaScript code (in this case ours) work.

Then we have the JavaScript code that allows us to *use* the functions made possible by d3.js.

```
var holder = d3.select("body") // select the 'body' element
      .append("svg")           // append an SVG element to the body
      .attr("width", 449)      // make the SVG element 449 pixels wide
      .attr("height", 249);    // make the SVG element 249 pixels high

// draw a circle
holder.append("circle")        // attach a circle
    .attr("cx", 200)           // position the x-center
    .attr("cy", 100)           // position the y-center
    .attr("r", 50);            // set the radius
```

I've broken the code into two separate portions to provide some clarity to their function. We could make it one block, but that wouldn't necessarily make it easier to understand.

Firstly we add a 'holder' for our graphics on the web page. I've named it `holder` but we could just as easily named it anything we wanted.

```
var holder = d3.select("body") // select the 'body' element
      .append("svg")           // append an SVG element to the body
      .attr("width", 449)      // make the SVG element 449 pixels wide
      .attr("height", 249);    // make the SVG element 249 pixels high
```

The first thing we do when declaring our holder is to select the `body` element of our web page (Remember those `<body>` tags in the HTML part earlier?).

Then we append a Scalable Vector Graphic (SVG) object to the body and we make it 449 pixels wide and 249 pixels high.

The `width` and `height` are 'attributes' of the SVG object. That is to say they describe a property of the object.

Believe it or not, I have made the container size unusual (not nice round numbers like 450 x 250) for a good reason. Later I will introduce a grid to our diagram so we can see where everything is laid out and this size makes the grid look better.

The second block of our JavaScript finally draws our circle.

```
holder.append("circle")        // attach a circle
    .attr("cx", 200)           // position the x-center
    .attr("cy", 100)           // position the y-center
    .attr("r", 50);            // set the radius
```

The first line appends a new element (a circle) to our SVG 'holder'.

If you like, you can think of having the `holder` declaration in front of the `.append("circle")` as being a nice short-hand way of writing the code. We could have had a much longer line that selected the `body`, appended the svg element and then appended our circle in one line, but it's actually a far better scheme for building multiple objects to break the sequences up which will allow us to manipulate groupings of objects in future code.

The second and third lines declare the attribute of our circle that specify where the centre of the circle is. In this case it's at the x/y position 200/100 (`cx`/`cy`).

The last line adds the radius attribute `r`. Here it is set to 50 pixels.

The three attributes `cx`, `cy` and `r` are all required when drawing a circle. There are other attributes we can put in there (and when we look at some of the upcoming elements, you should get a feel for them), but these are the minimum.

The purpose of describing this block of code that draws a circle isn't to show you how to draw a circle. This has only been a way of showing you how the code in the following sections is laid out and how it works. The elements we are going to generate can be drawn with exactly the same file but with just the section that adds the circle altered.

For example if you were to change this block of code;

```
holder.append("circle")        // attach a circle
    .attr("cx", 200)           // position the x-center
    .attr("cy", 100)           // position the y-center
    .attr("r", 50);            // set the radius
```

For this block of code;

```
holder.append("rect")          // attach a circle
    .attr("x", 150)            // x position of the top-left corner
    .attr("y", 50)             // y position of the top-left corner
    .attr("width", 100)        // set the rectangle width
    .attr("height", 100);      // set the rectangle height
```

Instead of drawing a circle we would be drawing a rectangle.

So this is what our circle will look like;

Circle

Because it will help a great deal to have a common frame of reference, I'm going to display the elements on a grid that looks a little like this;

Circle with Grid

The grid won't form part of the code that gets explained, but I will take the time out to describe how it's generated in another section, because it's quite cool in its own way :-).

With the grid in place it's far easier to see that the centre of our circle is indeed at the coordinates x = 200, y = 100 and that the radius is 50.

The circle is still somewhat plain, but bear with me because as we start to explore what we can do with styles and attributes we can add some variation to our elements.

Fancier Circle

With that explanation behind us we should begin our odyssey into the world of d3 elements.

Elements

We will begin by describing what we mean when we talk about an 'element'.

There is considerable scope for confusion when talking about elements on a web page. Are we talking about HTML elements[75], SVG elements[76] or something different?

In fact we are going to be describing a subset of SVG elements. Specifically those that are described in the d3.js API reference[77] (since that's why we're here right?). These are a collection of common shapes and objects which include circles, ellipses, rectangles, lines, polylines, polygons, text and paths.

"Text?" I hear you say. "Doesn't sound like a shape." I suppose it depends on how you think of it. We can use text in different ways in d3, but for this particular exercise we *can* regard text as an SVG element.

[75] http://reference.sitepoint.com/html/page-structure
[76] https://developer.mozilla.org/en-US/docs/Web/SVG/Element
[77] https://github.com/mbostock/d3/wiki/SVG-Shapes#svg-elements

Circle

A circle is a simple SVG shape[78] that is described by three required attributes.

- `cx`: The position of the centre of the circle in the x direction (left / right) measured from the left side of the screen.
- `cy`: The position of the centre of the circle in the y direction (up / down) measured from the top of the screen.
- `r`: The radius of the circle from the `cx`, `cy` position to the perimeter of the circle.

The following is an example of the code section required to draw a circle in conjunction with the HTML file outlined at the start of this chapter;

```
holder.append("circle")        // attach a circle
    .attr("cx", 200)           // position the x-center
    .attr("cy", 100)           // position the y-center
    .attr("r", 50);            // set the radius
```

This will produce a circle as follows;

Circle

The centre of the circle is at x = 200 and y = 100 and the radius is 50 pixels.

[78] https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-svg_circle

Ellipse

An ellipse[79] is described by four required attributes;

- `cx`: The position of the centre of the ellipse in the x direction (left / right) measured from the left side of the screen.
- `cy`: The position of the centre of the ellipse in the y direction (up / down) measured from the top of the screen.
- `rx`: The radius of the ellipse in the x dimension from the `cx`, `cy` position to the perimeter of the ellipse.
- `ry`: The radius of the ellipse in the y dimension from the `cx`, `cy` position to the perimeter of the ellipse.

The following is an example of the code section required to draw an ellipse in conjunction with the HTML file outlined at the start of this chapter;

```
holder.append("ellipse")        // attach an ellipse
    .attr("cx", 200)            // position the x-centre
    .attr("cy", 100)            // position the y-centre
    .attr("rx", 100)            // set the x radius
    .attr("ry", 50);            // set the y radius
```

This will produce an ellipse as follows;

Ellipse

The centre of the ellipse is at x = 200 and y = 100 and the radius is 50 pixels vertically and 100 pixels horizontally.

[79] https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-svg_ellipse

Rectangle

A rectangle[80] is described by four required attributes and two optional ones;

- `x`: The position on the x axis of the left hand side of the rectangle (required).
- `y`: The position on the y axis of the top of the rectangle (required).
- `width`: the width (in pixels) of the rectangle (required).
- `height`: the height (in pixels) of the rectangle (required).
- `rx`: The radius curve of the corner of the rectangle in the x dimension (optional).
- `ry`: The radius curve of the corner of the rectangle in the y dimension (optional).

The following is an example of the code section required to draw a rectangle (using only the required attributes) in conjunction with the HTML file outlined at the start of this chapter;

```
holder.append("rect")        // attach a rectangle
    .attr("x", 100)          // position the left of the rectangle
    .attr("y", 50)           // position the top of the rectangle
    .attr("height", 100)     // set the height
    .attr("width", 200);     // set the width
```

This will produce a rectangle as follows;

Rectangle

The top left corner of the rectangle is at 100, 50 and the rectangle is 200 pixels wide and 100 pixels high.

The following code section includes the optional attributes for the curved corners;

[80] https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-svg_rect

```
holder.append("rect")        // attach a rectangle
    .attr("x", 100)          // position the left of the rectangle
    .attr("y", 50)           // position the top of the rectangle
    .attr("height", 100)     // set the height
    .attr("width", 200)      // set the width
    .attr("rx", 10)          // set the x corner curve radius
    .attr("ry", 10);         // set the y corner curve radius
```

This will produce a rectangle (with curved corners) as follows;

Rectangle with curved corners

The corners are curved with radii in the x and y direction of 10 pixels.

Line

A line[81] is a simple line between two points and is described by four required attributes.

- `x1`: The x position of the first end of the line as measured from the left of the screen.
- `y1`: The y position of the first end of the line as measured from the top of the screen.
- `x2`: The x position of the second end of the line as measured from the left of the screen.
- `y2`: The y position of the second end of the line as measured from the top of the screen.

The following is an example of the code section required to draw a line in conjunction with the HTML file outlined at the start of this chapter. A notable addition to this code is the `style` declaration. In this case the line has no colour and this can be added with the `stroke` style which applies a colour to a line;

```
holder.append("line")          // attach a line
    .style("stroke", "black")  // colour the line
    .attr("x1", 100)     // x position of the first end of the line
    .attr("y1", 50)      // y position of the first end of the line
    .attr("x2", 300)     // x position of the second end of the line
    .attr("y2", 150);    // y position of the second end of the line
```

This will produce a line as follows;

Line

The line extends from the point 100,50 to 300,150.

[81] https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-svg_line

Polyline

A polyline[82] is a sequence of connected lines described with a single attribute. The d3.js wiki[83] rightly makes the point that "*it is typically more convenient and flexible to use the d3.svg.line path generator in conjunction with a path element*". So while drawing a polyline using this method may be possible, bear in mind that depending on your application, there may be a better way.

- `points`: The `points` attribute is a list of x,y coordinates that are the locations of the connecting points of the polyline.

The following is an example of the code section required to draw a polyline in conjunction with the HTML file outlined at the start of this chapter. A notable addition to this code are the `style` declarations. In this case the line of the polyline has no colour and this can be added with the `stroke` style which applies the colour black to a line. Likewise the area that is bounded by the polyline will be automatically filled with black unless we explicitly tell the object not to. This is achieved in this example by addition of the `fill` style to `none`.

```
holder.append("polyline")       // attach a polyline
    .style("stroke", "black")  // colour the line
    .style("fill", "none")     // remove any fill colour
    .attr("points", "100,50, 200,150, 300,50");  // x,y points
```

This will produce a polyline as follows;

Polyline

The polyline extends from the point 100,50 to 200,150 to 300,50.

[82] https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-svg_polyline
[83] https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-svg_polyline

Polygon

A polygon[84] is a sequence of connected lines which form a closed shape described with a single attribute. The d3.js wiki[85] rightly makes the point that "*it is typically more convenient and flexible to use the d3.svg.line path generator in conjunction with a path element*". So while drawing a polygon using this method may be possible, bear in mind that depending on your application, there may be a better way.

- `points`: The `points` attribute is a list of x,y coordinates that are the locations of the connecting points of the polygon. The last point is in turn connected to the first point.

The following is an example of the code section required to draw a polygon in conjunction with the HTML file outlined at the start of this chapter. A notable addition to this code are the `style` declarations. In this case the line of the polygon has no colour and this can be added with the `stroke` style which applies the colour black to a line. Likewise the area that is bounded by the polygon will be automatically filled with black unless we explicitly tell the object not to. This is achieved in this example by addition of the `fill` style to `none`.

```
holder.append("polygon")        // attach a polygon
    .style("stroke", "black")  // colour the line
    .style("fill", "none")     // remove any fill colour
    .attr("points", "100,50, 200,150, 300,50");  // x,y points
```

This will produce a polygon as follows;

Polyline

The polygon extends from the point 100,50 to 200,150 to 300,50 and then back to 100,50.

[84] https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-svg_polygon
[85] https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-svg_polygon

Path

A path[86] is an outline of an SVG shape which is described with a 'mini-language' inside a single attribute.

- `d`: This attribute is a list of instructions that allow a shape to be drawn in a complex way using a 'mini-language' of commands[87]. These commands are written in a shorthand of single letters such as `M`-moveto, `Z`-closepath, `L`-lineto, `C`-curveto. These commands can be absolute (normally designated by capital letters) or relative (lower case).

The following is an example of the code section required to draw a triangle in conjunction with the HTML file outlined at the start of this chapter. A notable addition to this code are the `style` declarations. In this case the line of the path has no colour and this can be added with the `stroke` style which applies the colour black to a line. Likewise the area that is bounded by the path will be automatically filled with black unless we explicitly tell the object not to. This is achieved in this example by addition of the `fill` style to `none`.

```
holder.append("path")          // attach a path
    .style("stroke", "black")  // colour the line
    .style("fill", "none")     // remove any fill colour
    .attr("d", "M 100,50, L 200,150, L 300,50 Z");  // path commands
```

This will produce a path as follows;

Path

The path mini-language first moves (`M`) to 100,50 then draws a line (`L`) to 200,150 then draws another line (`L`) to 300,50 then closes the path (`Z`).

[86] https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-svg_path
[87] http://www.w3.org/TR/SVG/paths.html#PathData

Clipped Path (AKA clipPath)

A clipPath[88] is the path of a SVG shape that can be used in combination with another shape to remove any parts of the combined shape that doesn't fall within the clipPath. That sounds slightly confusing, so we will break it down a bit to hopefully clarify the explanation.

Let's imagine that we want to display the intersection of two shapes. What we will do is define our `clipPath` which will act as a 'cookie cutter' which can cut out the shape we want (we will choose an ellipse). Then we will draw our base shape (which is analogous to the dough) that we will use our cookie cutter on (our dough will be shaped as a rectangle). The intersection of the cookie cutter and the dough is our clipped path.

Our clipPath (cookie cutter) element is an ellipse;

clipPath (cookie cutter)

Our shape that we will be clipping (the dough) is a rectangle;

Rectangle element (dough)

The intersection of the two is the clipped path (shaded grey);

[88] https://developer.mozilla.org/en-US/docs/Web/SVG/Element/clipPath

Combination of the ellipse and the rectangle

The graphic examples above are misleading in the sense that the two basic shapes are not actually displayed. All that results from the use of the `clipPath` is the region that is the intersection of the two.

The clipped path

The following is an example of the code section required to draw the clipped path in conjunction with the HTML file outlined at the start of this chapter. The `clipPath` element is given the `ID` 'ellipse-clip' and a specified size and location. Then when the rectangle is appended. the `clipPath` is specified as an attribute (via a URL) using `clip-path`.

```
// define the clipPath
holder.append("clipPath")       // define a clip path
        .attr("id", "ellipse-clip") // give the clipPath an ID
  .append("ellipse")            // shape it as an ellipse
        .attr("cx", 175)            // position the x-centre
        .attr("cy", 100)            // position the y-centre
        .attr("rx", 100)            // set the x radius
        .attr("ry", 50);            // set the y radius

// draw clipped path on the screen
holder.append("rect")       // attach a rectangle
    .attr("x", 125)         // position the left of the rectangle
    .attr("y", 75)          // position the top of the rectangle
    .attr("clip-path", "url(#ellipse-clip)") // clip the rectangle
    .style("fill", "lightgrey")   // fill the clipped path with grey
    .attr("height", 100)    // set the height
    .attr("width", 200);    // set the width
```

This will produce a path as follows;

The clipped path

An example of this in use can bee seen in the difference chart explanation later in the book.

Text

A text[89] element is an SVG object which is shaped as text. It is described by two required attributes and three optional ones.

- `x`: This attribute designates the anchor point location for the text in the x dimension (required).
- `y`: This attribute designates the anchor point location for the text in the y dimension (required).
- `dx`: This attribute designates the offset of the text from the anchor point in the x dimension (optional). There are several different sets of units that can be used to designated the offset of the text from an anchor point. These include `em` which is a scalable unit (used in these examples), `px` (pixels), `pt` (points (kind of like pixels)) and `5` (percent (scalable and kind of like `em`))
- `dy`: This attribute designates the offset of the text from the anchor point in the y dimension (optional).
- `text-anchor`: This attribute controls the horizontal text alignment (optional). It has three values; `start` (left aligned), `middle` (centre aligned) and `end` (right aligned).

The following is an example of the code section required to draw the text "Hello World" in conjunction with the HTML file outlined at the start of this chapter. A notable addition to this code is the `style` declaration which applies a `black fill` to the text. Additionally there is the declaration `.text` which defines the text that will be displayed.

```
holder.append("text")          // append text
    .style("fill", "black")    // fill the text with the colour black
    .attr("x", 200)            // set x position of left side of text
    .attr("y", 100)            // set y position of bottom of text
    .text("Hello World");      // define the text to display
```

This will produce text as follows;

Text

It can be seen from the image that the anchor point for the text is at 200,100 and that the text is positioned with this anchor point at the bottom, left of the text.

The following examples will demonstrate the various options for positioning and aligning text so that you can arrange it correctly.

Anchor at the bottom, middle of the text:

[89] https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-svg_text

```
holder.append("text")          // append text
    .style("fill", "black")    // fill the text with the colour black
    .attr("x", 200)            // set x position of left side of text
    .attr("y", 100)            // set y position of bottom of text
    .attr("text-anchor", "middle") // set anchor y justification
    .text("Hello World");          // define the text to display
```

This will produce text as follows;

Text: Anchored Bottom-middle

Anchor at the bottom, right of the text:

```
holder.append("text")          // append text
    .style("fill", "black")    // fill the text with the colour black
    .attr("x", 200)            // set x position of left side of text
    .attr("y", 100)            // set y position of bottom of text
    .attr("text-anchor", "end")  // set anchor y justification
    .text("Hello World");        // define the text to display
```

This will produce text as follows;

Text: Anchored Bottom-Right

Anchor at the middle, left of the text:

```
holder.append("text")          // append text
    .style("fill", "black")    // fill the text with the colour black
    .attr("x", 200)            // set x position of left side of text
    .attr("y", 100)            // set y position of bottom of text
    .attr("dy", ".35em")           // set offset y position
    .attr("text-anchor", "start")  // set anchor y justification
    .text("Hello World");          // define the text to display
```

This will produce text as follows;

Text: Anchored Middle-Left

Anchor in the middle, centre of the text:

```
holder.append("text")          // append text
    .style("fill", "black")    // fill the text with the colour black
    .attr("x", 200)            // set x position of left side of text
    .attr("y", 100)            // set y position of bottom of text
    .attr("dy", ".35em")           // set offset y position
    .attr("text-anchor", "middle") // set anchor y justification
    .text("Hello World");          // define the text to display
```

This will produce text as follows;

Text: Anchored Middle-Centre

Anchor in the middle, right of the text:

```
holder.append("text")          // append text
    .style("fill", "black")    // fill the text with the colour black
    .attr("x", 200)            // set x position of left side of text
    .attr("y", 100)            // set y position of bottom of text
    .attr("dy", ".35em")           // set offset y position
    .attr("text-anchor", "end")  // set anchor y justification
    .text("Hello World");          // define the text to display
```

This will produce text as follows;

Text: Anchored Middle-Right

Anchor at the top, left of the text:

```
holder.append("text")          // append text
    .style("fill", "black")    // fill the text with the colour black
    .attr("x", 200)            // set x position of left side of text
    .attr("y", 100)            // set y position of bottom of text
    .attr("dy", ".71em")           // set offset y position
    .attr("text-anchor", "start")  // set anchor y justification
    .text("Hello World");          // define the text to display
```

This will produce text as follows;

Text: Anchored Top-Left

Anchor at the top, middle of the text:

```
holder.append("text")          // append text
    .style("fill", "black")    // fill the text with the colour black
    .attr("x", 200)            // set x position of left side of text
    .attr("y", 100)            // set y position of bottom of text
    .attr("dy", ".71em")           // set offset y position
    .attr("text-anchor", "middle")  // set anchor y justification
    .text("Hello World");          // define the text to display
```

This will produce text as follows;

Text: Anchored Top-Middle

Anchor at the top, right of the text:

```
holder.append("text")          // append text
    .style("fill", "black")    // fill the text with the colour black
    .attr("x", 200)            // set x position of left side of text
    .attr("y", 100)            // set y position of bottom of text
    .attr("dy", ".71em")           // set offset y position
    .attr("text-anchor", "end")    // set anchor y justification
    .text("Hello World");          // define the text to display
```

This will produce text as follows;

Text: Anchored Top-Right

Attributes

At the start of writing this section I was faced with the question "What's an attribute?". But a reasonable answer has eluded me, so I will make the assumption that the answer will be something of a compromise :-). I like to think that an attribute of an element is something that is a characteristic of the object without defining it, and/or it may affect the object's position or orientation on the page. There could be a strong argument to say that the following section on styles could be seen to cross-over into attributes and I agree. However, for the purposes of providing a description of the syntax and effects, I'm happy with the following list :-).

Because not all attributes are applicable to all elements, there will be a bit of variation in the type of shapes we deal with in the description below, but there won't be any that are different to those that we've already looked at. There will be some repetition with recurring information from the elements section. This is intentional to hopefully allow each section to exist in its own right.

x, y

The `x` and `y` attributes are used to designate a position on the web page that is set from the top, left hand corner of the web page. Using the `x` and `y` attributes places the anchor points for these elements at a specified location. Of the elements that we have examined thus far, the rectangle element and the text element have anchor points to allow them to be positioned.

For example the following is a code section required to draw a rectangle (using only the required attributes) in conjunction with the HTML file outlined at the start of this chapter;

```
holder.append("rect")        // attach a rectangle
    .attr("x", 100)          // position the left of the rectangle
    .attr("y", 50)           // position the top of the rectangle
    .attr("height", 100)     // set the height
    .attr("width", 200);     // set the width
```

This will produce a rectangle as follows;

Rectangle with x,y at 100,50

The top left corner of the rectangle is specified using `x` and `y` at 100 and 50 respectively.

x1, x2, y1, y2

The `x1`, `x2`, `y1` and `y2` attributes are used to designate the position of two points on a web page that are set from the top, left hand corner of the web page. These two points are connected with a line as part of the `line` element.

The attributes are described as follows;

- `x1`: The x position of the first end of the line as measured from the left of the screen.
- `y1`: The y position of the first end of the line as measured from the top of the screen.
- `x2`: The x position of the second end of the line as measured from the left of the screen.
- `y2`: The y position of the second end of the line as measured from the top of the screen.

The following is an example of the code section required to draw a line in conjunction with the HTML file outlined at the start of this chapter. The attributes connect the point 100,50 (`x1`, `y1`) with 300,150 (`x2`, `y2`);

```
holder.append("line")          // attach a line
    .style("stroke", "black")  // colour the line
    .attr("x1", 100)     // x1 position of the first end of the line
    .attr("y1", 50)      // y1 position of the first end of the line
    .attr("x2", 300)     // x2 position of the second end of the line
    .attr("y2", 150);    // y2 position of the second end of the line
```

This will produce a line as follows;

Line

The line extends from the point 100,50 to 300,150.

points

The `points` attribute is used to set a series of points which are subsequently connected with a line and / or which may form the bounds of a shape. These are specifically associated with the `polyline` and `polygon` elements. Like the `x`, `y` and `x1`, `x2`, `y1`, `y2` attributes, the coordinates are set from the top, left hand corner of the web page.

The data for the points is entered as a sequence of x,y points in the following format;

```
.attr("points", "100,50, 200,150, 300,50");
```

Where 100,50 is the first x,y point then 200,150 is the second. Now is probably the best time to mention that the d3.js wiki[90] makes the point that "*it is typically more convenient and flexible to use the d3.svg.line path generator in conjunction with a path element*" when describing complex shapes. So while drawing a polyline or polygon using this method may be possible, bear in mind that depending on your application, there may be a better way.

The following is an example of the code section required to draw a polyline in conjunction with the HTML file outlined at the start of this chapter. The additional `style` declarations are included to illustrate the shape better. The `points` values can be compared with the subsequent image.

```
holder.append("polyline")      // attach a polyline
    .style("stroke", "black")  // colour the line
    .style("fill", "none")     // remove any fill colour
    .attr("points", "100,50, 200,150, 300,50");  // x,y points
```

This will produce a polyline as follows;

Polyline using `points` attribute

The polyline extends from the point 100,50 to 200,150 to 300,50.

[90] https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-svg_polyline

cx, cy

The cx, cy attributes are associated with the circle and ellipse elements and designate the centre of each shape. The coordinates are set from the top, left hand corner of the web page.

- cx: The position of the centre of the element in the x axis measured from the left side of the screen.
- cy: The position of the centre of the element in the y axis measured from the top of the screen.

The following is an example of the code section required to draw an ellipse in conjunction with the HTML file outlined at the start of this chapter. In it the centre of the ellipse is set by cx, cy as 200, 100.

```
holder.append("ellipse")        // attach an ellipse
    .attr("cx", 200)            // position the x-centre
    .attr("cy", 100)            // position the y-centre
    .attr("rx", 100)            // set the x radius
    .attr("ry", 50);            // set the y radius
```

This will produce an ellipse as follows;

Ellipse with Centre at 200, 100

The centre of the ellipse is at x = 200 and y = 100 and the radius is 50 pixels vertically and 100 pixels horizontally.

r

The `r` attribute determines the radius of a circle element from the `cx`, `cy` position (the centre of the circle) to the perimeter of the circle.

The following is an example of the code section required to draw a circle in conjunction with the HTML file outlined at the start of this chapter;

```
holder.append("circle")        // attach a circle
    .attr("cx", 200)           // position the x-center
    .attr("cy", 100)           // position the y-center
    .attr("r", 50);            // set the radius
```

This will produce a circle with a radius of 50 pixels as follows;

Circle with Radius of 50 Pixels

The centre of the circle is at x = 200 and y = 100 and the radius is 50 pixels.

rx, ry

The rx, ry attributes are associated with the ellipse element and designates the radius in the x direction (rx) and the radius in the y direction (ry).

- rx: The radius of the ellipse in the x dimension from the cx, cy position to the perimeter of the ellipse.
- ry: The radius of the ellipse in the y dimension from the cx, cy position to the perimeter of the ellipse.

The following is an example of the code section required to draw an ellipse in conjunction with the HTML file outlined at the start of this chapter. In it, the centre of the ellipse is set by cx, cy as 200, 100 and the radius in the x direction (rx) is 100 pixels and the radius in the y direction (ry) is 50 pixels.

```
holder.append("ellipse")        // attach an ellipse
    .attr("cx", 200)            // position the x-centre
    .attr("cy", 100)            // position the y-centre
    .attr("rx", 100)            // set the x radius
    .attr("ry", 50);            // set the y radius
```

This will produce an ellipse as follows;

Ellipse with x Radius of 100 and y Radius of 50

The centre of the ellipse is at x = 200 and y = 100 and the radius is 50 pixels vertically and 100 pixels horizontally.

transform (translate(x,y), scale(k), rotate(a))

The transform attribute is a powerful one which allows us to change the properties of an element in several different ways.

- `translate`: Where the element is moved by a relative value in the x,y direction.
- `scale`: Where the element's attributes are increased or reduced by a specified factor.
- `rotate`: Where the element is rotated about its reference point by an angular value.

Without a degree of prior understanding, these transforms can appear to behave in unusual ways, but hopefully we'll explain it sufficiently here so that you can appreciate the logic in the way they work.

transform (translate(x,y))

The transform-translate attribute will take an elements position and adjust it based on a specified value(s) in the x,y directions.

The best way to illustrate this is with an example;

This is the code snippet from the HTML file outlined at the start of this chapter which draws a circle at the position 200,100 (cx,cy);

```
holder.append("circle")        // attach a circle
    .attr("cx", 200)           // position the x-center
    .attr("cy", 100)           // position the y-center
    .attr("r", 50);            // set the radius
```

This will produce a circle as follows;

Circle

If we add in a `transform (translate(*x*,*y*))` attribute for values of x,y of 50,50 this will shift our circle by an additional 50 pixels in the x direction and 50 pixels in the y direction.

Here's the code snippet that will draw our new circle;

```
holder.append("circle")        // attach a circle
    .attr("cx", 200)           // position the x-center
    .attr("cy", 100)           // position the y-center
    .attr("transform", "translate(50,50)") // translate the circle
    .attr("r", 50);            // set the radius
```

And here's the resulting change;

Circle

The circle was positioned at the point 200,100 and then translated by 50 pixels in both axes to 250,150.

The original code snippet could in fact be written as follows;

```
holder.append("circle")        // attach a circle
    .attr("transform", "translate(200,100)") // translate the circle
    .attr("r", 50);            // set the radius
```

Since by default our starting position is 0,0 if we apply a translation of 200,100 we will end up at 200,100.

`transform (scale(k))`

The translate-scale attribute will take an element's attributes and scale them by a factor *k*.

Originally I thought that this attribute would affect the size of the element, but it affects more than that! As with the transform-translate attribute, the best way to illustrate this is with an example;

The following code snippet (in conjunction with the HTML file outlined at the start of this chapter) which draws a circle at the position 150,50 with a radius of 25 pixels;

```
holder.append("circle")     // attach a circle
    .attr("cx", 150)        // position the x-centre
    .attr("cy", 50)         // position the y-centre
    .attr("r", 25);         // set the radius
```

This will produce a circle as follows;

Circle

If we now introduce a transform-scale attribute with a scale of 2 we will see all three of the other attributes (`cx`, `cy` and `r`) scaled by a factor of two to 300, 100 and 50 respectively.

Here is the code;

```
holder.append("circle")              // attach a circle
    .attr("cx", 150)                 // position the x-centre
    .attr("cy", 50)                  // position the y-centre
    .attr("r", 25)                   // set the radius
    .attr("transform", "scale(2)"); // scale the circle attributes
```

Which will produce a circle as follows;

Circle

In this example we can see that the position (`cx`, `cy`) and the radius (`r`) have been scaled up by a factor of 2.

`transform (rotate(a))`

The translate-rotate attribute will rotate an element and its attributes by a declared angle in degrees.

The ability to rotate elements is obviously a valuable tool. The transform-rotate attribute does a great job of it, but the key to making sure that you know exactly what will happen to an object is to remember where the anchor point is for the object and to ensure that the associated attributes

are set appropriately. As with the transform translate & scale attributes, the best way to illustrate this is with an example;

The following is the code snippet (in conjunction with the HTML file outlined at the start of this chapter) which draws the text "Hello World" at the position 200,100 with the anchor point being the the middle of the text;

```
holder.append("text")         // append text
    .style("fill", "black")   // fill the text with the colour black
    .attr("x", 200)           // set x position of left side of text
    .attr("y", 100)           // set y position of bottom of text
    .attr("dy", ".35em")           // set offset y position
    .attr("text-anchor", "middle") // set anchor y justification
    .text("Hello World");          // define the text to display
```

This will produce text as follows;

Text: Anchored Middle-Centre

If we then apply a transform-rotate of 10 degrees as follows;

```
holder.append("text")         // append text
    .style("fill", "black")   // fill the text with the colour black
    .attr("x", 200)           // set x position of left side of text
    .attr("y", 100)           // set y position of bottom of text
    .attr("dy", ".35em")           // set offset y position
    .attr("text-anchor", "middle") // set anchor y justification
    .attr("transform", "rotate(10)")
    .text("Hello World");          // define the text to display
```

We will see the following on the screen;

Text: Anchored Middle-Centre

Obviously the text has been rotated, but hopefully you'll have noticed that it's also been displaced. This is because the transform-rotate attribute has been applied to both the text element (which *has* been rotated by 10 degrees) *and* the x,y attributes. If you imagine the origin point for the element being at 0,0, the centre, middle of the text element has been rotated about the point 0,0 by 10 degrees (hopefully slightly better explained in the following picture).

Text: All positioning Attributes Rotated

This could be seen as an impediment to getting things to move / change as you want to, but instead it's an indication of a different way of doing things. The solution to this particular feature is to combine the transform-rotate with the transform-translate that we used earlier so that the code looks like this;

```
holder.append("text")          // append text
    .style("fill", "black")    // fill the text with the colour black
    .attr("dy", ".35em")           // set offset y position
    .attr("text-anchor", "middle") // set anchor y justification
    .attr("transform", "translate(200,100) rotate(10)")
    .text("Hello World");          // define the text to display
```

And the image on the page looks like this;

Text: Rotated by 10 Degrees Anchored Middle-Centre

Which leads us to the final example for which is a combination of all three aspects of the transform attribute.

```
holder.append("text")          // append text
    .style("fill", "black")    // fill the text with the colour black
    .attr("dy", ".35em")           // set offset y position
    .attr("text-anchor", "middle") // set anchor y justification
    .attr("transform", "translate(200,100) scale(2) rotate(10)")
    .text("Hello World");          // define the text to display
```

Text: Translated, Scaled and Rotated

Here we have a text element translated to its position on the page, rotated by 10 degrees about the centre of the text and scaled by a factor of two.

width, height

`width` and `height` are required attributes of the rectangle element. `width` designates the width of the rectangle and `height` designates the height (If you're wondering, I often struggle defining the obvious).

The following is an example of the code section required to draw a rectangle (using only the required attributes) in conjunction with the HTML file outlined at the start of this chapter;

```
holder.append("rect")        // attach a rectangle
    .attr("x", 100)          // position the left of the rectangle
    .attr("y", 50)           // position the top of the rectangle
    .attr("height", 100)     // set the height
    .attr("width", 200);     // set the width
```

This will produce a rectangle as follows;

Rectangle

The width of the triangle is 200 pixels and the height is 100 pixels.

text-anchor

The `text-anchor` attribute determines the justification of a text element

Text can have one of three `text-anchor` types;

- `start` where the text is left justified.
- `middle` where the text is centre justified.
- `end` where the text is right justified.

The following is an example of code that will draw three separate lines of text with the three different `text-anchor` types in conjunction with the HTML file outlined at the start of this chapter;

```
holder.append("text")         // append text
    .style("fill", "black")   // fill the text with the colour black
    .attr("x", 200)           // set x position of left side of text
    .attr("y", 50)            // set y position of bottom of text
    .attr("text-anchor", "start") // set anchor y justification
    .text("Hello World - start"); // define the text to display

holder.append("text")         // append text
    .style("fill", "black")   // fill the text with the colour black
    .attr("x", 200)           // set x position of left side of text
    .attr("y", 100)           // set y position of bottom of text
    .attr("text-anchor", "middle") // set anchor y justification
    .text("Hello World - middle"); // define the text to display

holder.append("text")         // append text
    .style("fill", "black")   // fill the text with the colour black
    .attr("x", 200)           // set x position of left side of text
    .attr("y", 150)           // set y position of bottom of text
    .attr("text-anchor", "end") // set anchor y justification
    .text("Hello World - end"); // define the text to display
```

This will produce an output as follows;

Text with Different `text-anchor` Attributes

dx, dy

dx and dy are optional attributes that designate an offset of text elements from the anchor point in the x and y dimension . There are several different sets of units that can be used to designate the offset of the text from an anchor point. These include em which is a scalable unit, px (pixels), pt (points (kind of like pixels)) and % (percent (scalable and kind of like em))

We can demonstrate the offset effect by noting the difference in two examples.

The first is a simple projection of SVG text that aligns the text "Hello World" above and to the right of the anchor point at 200,100 (It does this in conjunction with the HTML file outlined at the start of this chapter.).

```
holder.append("text")          // append text
    .style("fill", "black")    // fill the text with the colour black
    .attr("x", 200)            // set x position of left side of text
    .attr("y", 100)            // set y position of bottom of text
    .text("Hello World");      // define the text to display
```

Which produces the following on the page;

Text with the Anchor at the Bottom Left Corner

The second example introduces the dx attribute setting the offset to 50 pixels. This adds another 50 pixels to the x dimension. We also introduce the dy attribute with an offset of .35em. This scalable unit allows the text to be set as a factor of the size of the text. In this case .35em will add half the height of the text to the y dimension placing the text so that it is exactly in the middle (vertically) of the 100 pixel line on the y dimension.

```
holder.append("text")          // append text
    .style("fill", "black")    // fill the text with the colour black
    .attr("x", 200)            // set x position of left side of text
    .attr("y", 100)            // set y position of bottom of text
    .attr("dx", "50px")        // set offset x position
    .attr("dy", ".35em")       // set offset y position
    .text("Hello World");      // define the text to display
```

Which produces the following on the page;

Text with 50 Pixel x Offset and Half Height y Offset

The text has been moved 50 pixels to the right and half the height of the text down the page.

textLength

The `textLength` attribute adjusts the length of the text to fit a specified value.

The following is a code snippet that prints the text "Hello World" above and to the right of the anchor point at 200,100 (It does this in conjunction with the HTML file outlined at the start of this chapter.). The addition of the `textLength` attribute declaration in the code stretches the "Hello World" out so that it fills 150 pixels.

```
holder.append("text")          // append text
    .style("fill", "black")    // fill the text with the colour black
    .attr("x", 200)            // set x position of left side of text
    .attr("y", 100)            // set y position of bottom of text
    .attr("textLength", "150") // set text length
    .text("Hello World");      // define the text to display
```

Which produces the following on the page;

Text Stretched to 150 Pixels Wide

It is worth noting that while the text has been spread out, the individual letters remain unstretched. Only the letter and word spacing has been adjusted. However, using the `lengthAdjust` attribute can change this.

lengthAdjust

The `lengthAdjust` attribute allows the `textLength` attribute to have the spacing of a text element controlled to be either `spacing` or `spacingAndGlyphs`;

- `spacing`: In this option the letters remain the same size, but the spacing between the letters and words are adjusted.
- `spacingAndGlyphs`: In this option the text is stretched or squeezed to fit.

The attribute can be best illustrated via an example. The following code snippet (which works in conjunction with the HTML file outlined at the start of this chapter) shows three versions of the text element. The top line is the standard text. The middle line is the `textLength` set to 150 and the `lengthAdjust` set to `spacing` (which is the default). The bottom line is the `textLength` set to 150 and the `lengthAdjust` set to `spacingAndGlyphs`.

```
holder.append("text")          // append text
    .style("fill", "black")    // fill the text with the colour black
    .attr("x", 200)            // set x position of left side of text
    .attr("y", 50)             // set y position of bottom of text
    .text("Hello World");      // define the text to display

holder.append("text")          // append text
    .style("fill", "black")    // fill the text with the colour black
    .attr("x", 200)            // set x position of left side of text
    .attr("y", 100)            // set y position of bottom of text
    .attr("textLength", "150") // set text length
    .attr("lengthAdjust", "spacing")
    .text("Hello World");      // define the text to display

holder.append("text")          // append text
    .style("fill", "black")    // fill the text with the colour black
    .attr("x", 200)            // set x position of left side of text
    .attr("y", 150)            // set y position of bottom of text
    .attr("textLength", "150") // set text length
    .attr("lengthAdjust", "spacingAndGlyphs")
    .text("Hello World");      // define the text to display
```

The image on the screen will look like the following;

Text Stretched in Three Ways

The image shows that the top line looks normal, the middle line has had the spaces increased to increase the length of the text and the bottom line has been stretched.

Styles

What's a style?

Believe it or not, that's as difficult a question to answer as "What's an attribute?". I like to think that an element can be selected and arranged on a web page with `select` and `attr`, but once it's there, changes to how it looks are a matter for `style`. We will cover a range of qualities that neatly fit into this definition in the following section (such as fill, opacity and stroke-width) but there are also a range of unusual style declarations that many may not have come across (I certainly hadn't before writing this).

The other important thing to mention about setting styles for elements is that there are different ways to accomplish the task. We'll go through the process of describing different styles as they can be applied to individual elements in isolation, but there is a more powerful way to manage styles across a range of elements via Cascading Style Sheets (CSS) in the `<style>` section of a web page or even via an external style sheet. We will examine these possibilities at the end of the section.

Full disclosure: I have not figured out how to work some of the styles for d3.js I'm afraid that `clip-path` and `mask` have exceeded my skill-set and I will have to leave them for another day :-(. I found that there are several good examples that make use of these styles, but I have struggled (unsuccessfully) to present them in a simple example.

fill

The `fill` style will fill the element being presented with a specified colour.

By default, most elements will be filled with black (the majority of the examples used in this chapter make no `fill` declaration).

The following example (which works in conjunction with the HTML file outlined at the start of this chapter) shows the syntax for filling a simple circle with the colour red;

```
holder.append("circle")        // attach a circle
    .attr("cx", 200)           // position the x-centre
    .attr("cy", 100)           // position the y-centre
    .attr("r", 50)             // set the radius
    .style("fill", "red");     // set the fill colour
```

Which results in the following image;

Circle with Red Fill

As we saw with the `polyline` and `polygon` examples earlier in the chapter some shapes may need to have their `fill` colour turned off in some circumstances and this can be accomplished by declaring the colour to be `none` (`.style("fill", "none");`).

There are several different ways to define exactly what colour we want as a fill. The example above uses a 'named colour code' to declare the colour as "red" but we could also have defined it as rgb (`.style("fill", "rgb(255,0,0)");`) or in hexadecimal (`.style("fill", "#f00");`)

stroke

The `stroke` style applies a colour to lines.

By default many elements do not have a stroke colour set, so it's a matter of declaring the colour with either a named colour code ("red"), an rgb value ("rgb(255,0,0)") or the appropriate hex ("#f00").

The following example (which works in conjunction with the HTML file outlined at the start of this chapter) shows the syntax for applying the colour red to a simple circle. The fill has been set to `none` to help the colour stand out.

```
holder.append("circle")        // attach a circle
    .attr("cx", 200)           // position the x-centre
    .attr("cy", 100)           // position the y-centre
    .attr("r", 50)             // set the radius
    .style("stroke", "red")    // set the line colour
    .style("fill", "none");    // set the fill colour
```

Which results in the following image;

Circle with Red Border

opacity

The `opacity` style has the effect of varying an element's transparency.

The valid range for `opacity` is from `0` (completely transparent) to `1` (solid colour). We should make the distinction at this point that `opacity` affects the entire element, whereas the following `fill-opacity` and `stroke-opacity` affects only the fill and stroke respectively.

The following code snippet (which works in conjunction with the HTML file outlined at the start of this chapter) creates a green circle with a red border. The opacity value of .2 creates a degree of transparency which will show the grid lines underneath the element.

```
holder.append("circle")        // attach a circle
    .attr("cx", 200)           // position the x-centre
    .attr("cy", 100)           // position the y-centre
    .attr("r", 50)             // set the radius
    .style("opacity", .2)      // set the element opacity
    .style("stroke", "red")    // set the line colour
    .style("fill", "green");   // set the fill colour
```

Which results in the following image;

Circle with opacity

fill-opacity

The `fill-opacity` style changes the transparency of the fill of an element.

The valid range for `fill-opacity` is from 0 (completely transparent) to 1 (solid colour). We should make the distinction at this point that `fill-opacity` affects only the fill of an element, whereas `opacity` will affect the entire element.

The following code snippet (which works in conjunction with the HTML file outlined at the start of this chapter) creates a green circle with a red border. The opacity value of .2 creates a degree of transparency for the fill which will show the grid lines underneath.

```
holder.append("circle")        // attach a circle
    .attr("cx", 200)           // position the x-centre
    .attr("cy", 100)           // position the y-centre
    .attr("r", 50)             // set the radius
    .style("fill-opacity", .2) // set the fill opacity
    .style("stroke", "red")    // set the line colour
    .style("fill", "green");   // set the fill colour
```

Which results in the following image;

Circle with Semi-Transparent Fill

The distinction between this image and the one for the `opacity` style clearly shows the line around the outside of the object as still a solid (opaque) colour.

stroke-opacity

The `stroke-opacity` style changes the transparency of the stroke (line) of an element.

The valid range for `stroke-opacity` is from `0` (completely transparent) to `1` (solid colour). We should make the distinction at this point that `stroke-opacity` affects only the line or border of an element, whereas `opacity` will affect the entire element.

The following code snippet (which works in conjunction with the HTML file outlined at the start of this chapter) creates an empty circle with a red border. The opacity value of .2 creates a degree of transparency for the stroke which will show the grid lines underneath (or at least make it appear more 'muted').

```
holder.append("circle")          // attach a circle
    .attr("cx", 200)             // position the x-centre
    .attr("cy", 100)             // position the y-centre
    .attr("r", 50)               // set the radius
    .style("stroke-opacity", .2) // set the stroke opacity
    .style("stroke", "red")      // set the line colour
    .style("fill", "none");      // set the fill colour
```

Which results in the following image;

Circle with Red Border and opacity

Although it is not necessarily easy to see in this example because the line is quite thin, the lines of the grid behind the circle will be showing through the line of the circle.

stroke-width

The `stroke-width` style adjusts the width of the line of an element.

The value specified when setting `stroke-width` is in pixels.

The following code snippet (which works in conjunction with the HTML file outlined at the start of this chapter) creates an empty circle with a red border. The `stroke-width` is set to 5 which equates to 5 pixels (it can also be specified as "5px").

```
holder.append("circle")          // attach a circle
    .attr("cx", 200)             // position the x-centre
    .attr("cy", 100)             // position the y-centre
    .attr("r", 50)               // set the radius
    .style("stroke-width", 5)    // set the stroke width
    .style("stroke", "red")      // set the line colour
    .style("fill", "none");      // set the fill colour
```

Which results in the following image;

Circle with Thicker Red Border

The width of the line that forms the border of the circle is now 5 pixels wide :-).

stroke-dasharray

The `stroke-dasharray` style allows us to form element lines with dashes instead of solid lines.

We have covered dashed lines in practical way in a previous section of the book ('Make a Dashed Line') but for the sake of completeness I will include dashed lines here as well.

We create a dashed line by specifying the length of a dash and then the length of a space. We can include a long list of dashes and spaces and once complete our line will simply repeat the pattern we have specified.

For example the following code snippet (which works in conjunction with the HTML file outlined at the start of this chapter) creates a line with a dash of 10 pixels followed by a space of 2 pixels;

```
holder.append("circle")       // attach a circle
    .attr("cx", 200)          // position the x-centre
    .attr("cy", 100)          // position the y-centre
    .attr("r", 50)            // set the radius
    .style("stroke-dasharray", ("10,3")) // make the stroke dashed
    .style("stroke", "red")   // set the line colour
    .style("fill", "none");   // set the fill colour
```

Which results in the following image;

Circle with Dashed Red Border

More complex combinations of dashes and spaces are possible as are complex animation sequences that leverage the ability to move objects along a path (these are certainly more advanced examples).

stroke-linecap

The `stroke-linecap` style allows control of the shape of the ends of lines in d3.js.

There are three shape options;

- `butt` where the line simply butts up to the starting or ending position and is cut off squarely.
- `round` where the line is rounded in proportion to its width.
- `square` where the line is squared off but extended in proportion to its width.

The following code snippet (which works in conjunction with the HTML file outlined at the start of this chapter) generates three lines showing each `stroke-linecap` style option. The top line uses `butt`. The middle line uses `round` and the bottom line uses `square`.

```
holder.append("line")                  // attach a line
    .style("stroke", "black")          // colour the line
    .style("stroke-width", 20)         // adjust line width
    .style("stroke-linecap", "butt")   // stroke-linecap type
    .attr("x1", 100)     // x position of the first end of the line
    .attr("y1", 50)      // y position of the first end of the line
    .attr("x2", 300)     // x position of the second end of the line
    .attr("y2", 50);     // y position of the second end of the line

holder.append("line")                   // attach a line
    .style("stroke", "black")           // colour the line
    .style("stroke-width", 20)          // adjust line width
    .style("stroke-linecap", "round")   // stroke-linecap type
    .attr("x1", 100)     // x position of the first end of the line
    .attr("y1", 100)     // y position of the first end of the line
    .attr("x2", 300)     // x position of the second end of the line
    .attr("y2", 100);    // y position of the second end of the line

holder.append("line")                    // attach a line
    .style("stroke", "black")            // colour the line
    .style("stroke-width", 20)           // adjust line width
    .style("stroke-linecap", "square")   // stroke-linecap type
    .attr("x1", 100)     // x position of the first end of the line
    .attr("y1", 150)     // y position of the first end of the line
    .attr("x2", 300)     // x position of the second end of the line
    .attr("y2", 150);    // y position of the second end of the line
```

Which results in the following image;

Three Lines with Different End Shapes

The shapes are quite distinct for each type and it is useful to note the degree to which the lines extend beyond their start and end points.

stroke-linejoin

The `stroke-linejoin` style specifies the shape of the join of two lines. This would be used on `path`, `polyline` and `polygon` elements (and possibly more).

There are three line join options;

- `miter` where the join is squared off as would be expected at the join of two lines.
- `round` where the outside portion of the join is rounded in proportion to its width.
- `bevel` where the join has a straight edged outer portion clipped off to provide a slightly more contoured effect while still being angular.

The following code snippet (which works in conjunction with the HTML file outlined at the start of this chapter) generates a poly line where the join has the connection shaped using the `stroke-linejoin` `round` style.

```
holder.append("polyline")       // attach a polyline
    .style("stroke", "black")   // colour the line
    .style("fill", "none")      // remove any fill colour
        .style("stroke-width", 20)  // colour the line
        .style("stroke-linejoin", "round")  // shape the line join
    .attr("points", "100,50, 200,150, 300,50");  // x,y points
```

Which results in the following image;

Polyline with Round Join

Note the curve on the outer of the join.

Changing the shape of the line join to `bevel` produces the following;

Polyline with Bevel Join

Here we can see the clipping of the outer portion of the join.

And using `miter` produces a standard connection;

Polyline with Miter Join

This is the default setting for line joins and does not need to be added unless the line join type has already been set to a different default.

writing-mode

The `writing-mode` style changes the orientation of the text so that it prints out top to bottom. It has a single option "tb" that accomplishes this. It is relatively limited in scope compared to the equivalent for CSS, but for the purposes of generating some text it has a definite use.

The following code snippet (hich works in conjunction with the HTML file outlined at the start of this chapter) creates a line of text that is now printed from top to bottom instead of left to right.

```
holder.append("text")            // append text
    .style("fill", "black")      // make the text black
    .style("writing-mode", "tb") // set the writing mode
    .attr("x", 200)         // set x position of left side of text
    .attr("y", 100)         // set y position of bottom of text
    .text("Hello World");   // define the text to display
```

Which results in the following image;

Text rotated using writing-mode

It is significant to note that while it looks like the text has been rotated about it's anchor point, this actually isn't the case since the anchor point should be at 200,100. Also, the `glyph-orientation-vertical` style (which follows) will allow the text to be orientated vertically which will be useful.

glyph-orientation-vertical

The `glyph-orientation-vertical` style changes the rotation of the individual glyphs (characters) in text and if used in conjunction with the `writing-mode` style (and set to `0`) will allow the text to be displayed vertically with the letters orientated vertically as well.

The following code snippet (which works in conjunction with the HTML file outlined at the start of this chapter) creates a line of text that is now printed from top to bottom with letters orientated vertically.

```
holder.append("text")            // append text
    .style("fill", "black")      // make the text black
    .style("writing-mode", "tb") // set the writing mode
    .style("glyph-orientation-vertical", 0)
    .attr("x", 200)         // set x position of left side of text
    .attr("y", 25)          // set y position of bottom of text
    .text("Hello World");   // define the text to display
```

Which results in the following image;

Text rotated and orientated

It is worth noting that the text spacing increases dramatically as the spacing for each letter relies on the normal distance between the bottom and top of a line of text.

Using styles in Cascading Style Sheets

Declaring styles on an element by element basis is an OK way to apply styles, but when our visualizations become more complex, this can be an inefficient use of code.

A smarter way to provide a common set of styles to elements is to declare them in the `<style>` section of our HTML document using Cascading Style Sheets (CSS). These will then be automatically applied to our elements.

We start with an example script that draws our three lines that have different styles of linecaps. Our previous example looked like the following (in conjunction with the HTML file outlined at the start of this chapter)

```
holder.append("line")                // attach a line
    .style("stroke", "black")        // colour the line
    .style("stroke-width", 20)       // adjust line width
    .style("stroke-linecap", "butt") // stroke-linecap type
    .attr("x1", 100)     // x position of the first end of the line
    .attr("y1", 50)      // y position of the first end of the line
    .attr("x2", 300)     // x position of the second end of the line
    .attr("y2", 50);     // y position of the second end of the line

holder.append("line")                 // attach a line
    .style("stroke", "black")         // colour the line
    .style("stroke-width", 20)        // adjust line width
    .style("stroke-linecap", "round") // stroke-linecap type
    .attr("x1", 100)     // x position of the first end of the line
    .attr("y1", 100)     // y position of the first end of the line
    .attr("x2", 300)     // x position of the second end of the line
    .attr("y2", 100);    // y position of the second end of the line

holder.append("line")                  // attach a line
    .style("stroke", "black")          // colour the line
    .style("stroke-width", 20)         // adjust line width
    .style("stroke-linecap", "square") // stroke-linecap type
    .attr("x1", 100)     // x position of the first end of the line
    .attr("y1", 150)     // y position of the first end of the line
    .attr("x2", 300)     // x position of the second end of the line
    .attr("y2", 150);    // y position of the second end of the line
```

Which resulted in the following image;

Three Lines with Different End Shapes

The block of code for each of the three lines contains three separate `style` declarations. Two of which are identical for all three blocks of code;

```
    .style("stroke", "black")         // colour the line
    .style("stroke-width", 20)        // adjust line width
```

To make these styles available from a common point, we declare them in the `<style>` section of our HTML file as follows;

```
<style>
line.linecap {
  stroke: black;
  stroke-width: 20;
}
</style>
```

The `<style>` tags simply tell our browser which part of the HTML file we are using to define our styles.

The `line.linecap` portion identifies the following styles as belonging to the `line` elements that are also identified as belonging to the 'class' `linecap` (We have used the `linecap` name as a convenience only and it could just as easily been `foobar`.).

The two styles are enclosed within curly braces and are declared in the form `<style-name>: <style-value>;`. So for our example here, the stroke is black and its width is 20 pixels.

Then our example script can have the two styles removed from each of the blocks that draws the lines and in their place we add a new attribute `class` that assigns a class to the element (in this case the class `linecap`). Our new code will look like this;

```
holder.append("line")           // attach a line
    .style("stroke-linecap", "butt")  // stroke-linecap type
    .attr("class", "linecap")   // inherits styles from CSS
    .attr("x1", 100)     // x position of the first end of the line
    .attr("y1", 50)      // y position of the first end of the line
    .attr("x2", 300)     // x position of the second end of the line
    .attr("y2", 50);     // y position of the second end of the line

holder.append("line")           // attach a line
    .style("stroke-linecap", "round")  // stroke-linecap type
    .attr("class", "linecap")   // inherits styles from CSS
    .attr("x1", 100)     // x position of the first end of the line
    .attr("y1", 100)     // y position of the first end of the line
    .attr("x2", 300)     // x position of the second end of the line
    .attr("y2", 100);    // y position of the second end of the line

holder.append("line")           // attach a line
    .style("stroke-linecap", "square")  // stroke-linecap type
    .attr("class", "linecap")   // inherits styles from CSS
    .attr("x1", 100)     // x position of the first end of the line
    .attr("y1", 150)     // y position of the first end of the line
```

```
    .attr("x2", 300)     // x position of the second end of the line
    .attr("y2", 150);    // y position of the second end of the line
```

While this has only replaced two lines with one in our code, the potential for use in far more complex examples should be obvious. There is significantly more detail that can be gone into with regard to CSS, but that would be beyond my meagre abilities.

Assorted Tips and Tricks

Change a line chart into a scatter plot

Confession time.

I didn't actually intend to add in a section with a scatter plot in it for its own sake because I thought it would be;

1. tricky
2. not useful
3. all of the above

I was wrong on all counts.

> I did want to have a scatter plot, because I wanted to display tool tips, but this is too neat to ignore. It was literally a 5 minute job, 3 minutes of which was taken up by going to the d3 gallery on the wiki[91] and ogling at the cool stuff there before snapping out of it and going to the scatter plot example[92].

All you need to do is take the simple graph example file and slot the following block in between the 'Add the valueline path' and the 'add the x axis' blocks.

```
svg.selectAll("dot")
    .data(data)
.enter().append("circle")
    .attr("r", 3.5)
    .attr("cx", function(d) { return x(d.date); })
    .attr("cy", function(d) { return y(d.close); });
```

And you will get...

[91] https://github.com/mbostock/d3/wiki/Gallery
[92] http://bl.ocks.org/3887118

A scatter plot! (with a line)

The full code for this graph can also be found on github[93] or in the code samples bundled with this book (simple-scatterplot.html and data.csv). A live example can be found on bl.ocks.org[94].

I deliberately put the dots *after* the line in the drawing section, because I thought they would look better, but you could put the block of code before the line drawing block to get the following effect;

A scatter plot with the line in front of the dots

(just trying to reinforce the concept that 'order' matters when drawing objects :-)).

You could of course just remove the line block all together...

A scatter plot without the line this time

But in my humble opinion it loses something.

So what do the individual lines in the scatter plot block of JavaScript do?

[93] https://gist.github.com/d3noob/38744a17f9c0141bcd04
[94] http://bl.ocks.org/d3noob/38744a17f9c0141bcd04

The first line (`svg.selectAll("dot")`) essentially provides a suitable grouping label for the svg circle elements that will be added. The next line associates the range of data that we have to the group of elements we are about to add in.

Then we add a circle for each data point (`.enter().append("circle")`) with a radius of 3.5 pixels (`.attr("r", 3.5)`) and appropriate x (`.attr("cx", function(d) { return x(d.date); })`) and y (`.attr("cy", function(d) { return y(d.close); });`) coordinates.

There is lots more that we could be doing with this piece of code (check out the scatter plot example[95]) including varying the colour or size or opacity of the circles depending on the data and all sorts of really neat things, but for the mean time, there we go. Scatter plot!

[95] http://bl.ocks.org/3887118

Adding tooltips.

Tooltips have a marvellous duality. They are on one hand a pretty darned useful thing that aids in giving context and information where required and on the other hand, if done with a bit of care, they can look very stylish :-).

Technically, they represent a slight move from what we have been playing with so far into a mildly more complex arena of 'transitions' and 'events'. You can take this one of two ways. Either accept that it just works and implement it as shown, or you will know what's going on and feel free to deride my efforts as those of a rank amateur :-).

The source for the implementation was taken from Mike Bostock's example on bl.ocks.org[96]. This was combined with a few other bit's and pieces (the trickiest being working out how to format the displayed date correctly and inserting a line break in the tooltip (which I found on Google Groups[97]; (well done to all those participating in that discussion)). I make the assumption that any or all errors that occur in the implementation will be mine, whereas, any successes will be down to the original contributors.

Just in case there is some confusion, a tooltip (one word or two?) is a discrete piece of information that will pop into view when the mouse hovers over somewhere specific. Most of us have seen and used them, but I suppose we all tend to call them different things such as 'infotip', 'hint' or 'hover box' I don't know if there's a right name for them, but here's an example of what we're trying to achieve;

A tooltip magically appears over a dot

You can see the mouse has hovered over one of the scatter plot circles and a tip has appeared that provides the user with the exact date and value for that point.

Now, you may also notice that there's a certain degree of 'fancy' here as the information is bound by a rectangular shape with rounded corners and a slight opacity. The other piece of 'fancy' which you don't see in a PDF (or whatever format this distinguished tome will be published in on its 33rd reprint in the year 2034), is that when these tool tips appear and disappear, they do so in an elegant fade-in, fade-out way. Pretty!

Now, before we get started describing how the code goes together, let's take a quick look at the two technique specifics that I mentioned earlier, 'transitions' and 'events'.

Transitions

From the main d3.js web page (d3js.org) transitions are described as gradually interpolating styles and attributes over time. So what I take that to mean is that if you want to change an object, you

[96] http://bl.ocks.org/1087001
[97] https://groups.google.com/forum/?fromgroups=#!topic/d3-js/GgFTf24ltjc

can do so be simply specifying the attribute / style end point that you want it to end up with and the time you want it to take and go!

Of course, it's not quite that simple, but luckily, smarter people than I have done some fantastic work describing different aspects of transitions so please see the following for a more complete description of the topic;

- Mike Bostock's Bar chart tutorial[98]
- Christophe Viau's 'Try D3 Now!' tutorial[99]

Hopefully observing the mouseover and mouseout transitions in the tooltips example will whet your appetite for more!

Events

The other technique is related to mouse 'events'. This describes the browser watching for when 'something' happens with the mouse on the screen and when it does, it takes a specified action. A (probably non-comprehensive) list of the types of events are the following;

- mousedown: Triggered by an element when a mouse button is pressed down over it
- mouseup: Triggered by an element when a mouse button is released over it
- mouseover: Triggered by an element when the mouse comes over it
- mouseout: Triggered by an element when the mouse goes out of it
- mousemove: Triggered by an element on every mouse move over it.
- click: Triggered by a mouse click: mousedown and then mouseup over an element
- contextmenu: Triggered by a right-button mouse click over an element.
- dblclick: Triggered by two clicks within a short time over an element

How many of these are valid to use within d3 I'm not sure, but I'm willing to bet that there are probably more than those here as well. Please go to http://javascript.info/tutorial/mouse-events[100] for a far better description of the topic if required.

Get tipping

So, bolstered with a couple of new concepts to consider, let's see how they are enacted in practice.

The full code for this graph can also be found on github[101] or in the code samples bundled with this book (simple-tooltips.html and data.csv). A live example can be found on bl.ocks.org[102].

If we start with our simple-scatter plot graph there are 4 areas in it that we will want to modify (it may be easier to check the tooltips.html file in the example files in the downloads section on d3noob.org).

The first area is the CSS. The following code should be added just before the `</style>` tag;

[98] http://mbostock.github.com/d3/tutorial/bar-2.html
[99] http://christopheviau.com/d3_tutorial/
[100] http://javascript.info/tutorial/mouse-events
[101] https://gist.github.com/d3noob/a22c42db65eb00d4e369
[102] http://bl.ocks.org/d3noob/a22c42db65eb00d4e369

```
div.tooltip {
    position: absolute;
    text-align: center;
    width: 60px;
    height: 28px;
    padding: 2px;
    font: 12px sans-serif;
    background: lightsteelblue;
    border: 0px;
    border-radius: 8px;
    pointer-events: none;
}
```

These styles are defining how our tooltip will appear . Most of them are fairly straight forward. The position of the tooltip is done in absolute measurements, not relative. The text is centre aligned, the height, width and colour of the rectangle is 28px, 60px and lightsteelblue respectively. The 'padding' is an interesting feature that provides a neat way to grow a shape by a fixed amount from a specified size.

We set the border to 0px so that it doesn't show up and a neat style (attribute?) called border-radius provides the nice rounded corners on the rectangle.

Lastly, but by no means least, the 'pointer-events: none' line is in place to instruct the mouse event to go "through" the element and target whatever is "underneath" that element instead (Read more here[103]). That means that even if the tooltip partly obscures the circle, the code will still act as if the mouse is over only the circle.

The second addition is a simple one-liner that should (for forms sake) be placed under the 'parseData' variable declaration;

```
var formatTime = d3.time.format("%e %B");
```

This line formats the date when it appears in our tooltip. Without it, the time would default to a disturbingly long combination of temporal details. In the case here we have declared that we want to see the day of the month (%e) and the full month name(%B).

The third block of code is the function declaration for 'div'.

```
var div = d3.select("body").append("div")
    .attr("class", "tooltip")
    .style("opacity", 0);
```

We can place that just after the 'valueline' definition in the JavaScript. Again there's not too much here that's surprising. We tell it to attach 'div' to the body element, we set the class to the tooltip class (from the CSS) and we set the opacity to zero. It might sound strange to have the opacity set to zero, but remember, that's the natural state of a tooltip. It will live unseen until it's moment of revelation arrives and it pops up!

[103] https://developer.mozilla.org/en-US/docs/CSS/pointer-events

The final block of code is slightly more complex and could be described as a mutant version of the neat little bit of code that we used to do the drawing of the dots for the scatter plot. That's because the tooltips are all about the scatter plot circles. Without a circle to 'mouseover', the tooltip never appears :-).

So here's the code that includes the scatter plot drawing (it's included since it's pretty much integral);

```
svg.selectAll("dot")
    .data(data)
.enter().append("circle")
    .attr("r", 5)
    .attr("cx", function(d) { return x(d.date); })
    .attr("cy", function(d) { return y(d.close); })
    .on("mouseover", function(d) {
        div.transition()
            .duration(200)
            .style("opacity", .9);
        div        .html(formatTime(d.date) + "<br/>"  + d.close)
            .style("left", (d3.event.pageX) + "px")
            .style("top", (d3.event.pageY - 28) + "px");
        })
    .on("mouseout", function(d) {
        div.transition()
            .duration(500)
            .style("opacity", 0);
    });
```

Before we start going through the code, the example file for tooltips that is on d3noob.org includes a brief series of comments for the lines that are added or changed from the scatter plot, so if you want to compare what is going on in context, that is an option.

The first six lines of the code are a repeat of the scatter plot drawing script. The only changes are that we've increased the radius of the circle from 3.5 to 5 (just to make it easier to mouse over the object) and we've removed the semicolon from the `cy` attribute line since the code now has to carry on.

So the additions are broken into two areas that correspond to the two events. `mouseover` and `mouseout`. When the mouse moves over any of the circles in the scatter plot, the mouseover code is executed on the `div` element. When the mouse is moved off the circle a different set of instructions are executed.

There is only one!

It would be a mistake to think of tooltips in the plural because there aren't a whole series of individual tooltips just waiting to appear for their specific circle. There is only one tooltip that will appear when the mouse moves over a circle. And depending on what circle it's over, the properties of the tooltip will alter slightly (in terms of its position and contents).

on.mouseover

The `.on("mouseover"` line initiates the introduction of the tooltip. Then we declare the element we will be introducing ('`div`') and that we will be applying a transition to its introduction (`.transition()`). The next two lines describe the transition. It will take 200 milliseconds (`.duration(200)`) and will result in changing the element's opacity to .9 (`.style("opacity", .9);`). Given that the natural state of our tooltip is an opacity of 0, this make sense for something appearing, but it doesn't go all the way to a solid object and it retains a slight transparency just to make it look less permanent.

The following three lines format our tooltip. The first one adds an html element that contains our x and y information (the `date` and the `d.close` value). Now this is done in a slightly strange way. Other tooltips that I have seen have used a '`.text`' element instead of a '`.html`' one, but I have used '`.html`' in this case because I wanted to include the line break tag `<br/>` to separate the date and value. I'm sure there are other ways to do it, but this worked for me. The other interesting part of this line is that this is where we call our time formatting function that we described earlier. The next two lines position the tooltip on the screen and to do this they grab the x and y coordinates of the mouse when the event takes place (with the `d3.event.pageX` and `d3.event.pageY` snippets) and apply a correction in the case of the y coordinate to raise the tooltip up by the same amount as its height (28 pixels).

on.mouseout

The `.on("mouseout"` section is slightly simpler in that it doesn't have to do any fancy text / html / coordinate stuff. All it has to do is to fade out the 'div' element. And that is done by simply reversing the opacity back to 0 and setting the duration for the transition to 500 milliseconds (being slightly longer than the fade-in makes it look slightly cooler IMHO).

Right, there you go. As a description it's ended up being a bit of a wall of text I'm afraid. But hopefully between the explanation and the example code you will get the idea. Please take the time to fiddle with the settings described here to find the ones that work for you and in the process you will reinforce some of the principles that help D3 do its thing.

Including an HTML link in a tool tip

There was an interesting question on d3noob.org[104] about adding an HTML link to a tooltip. While the person asking the question had the problem pretty much solved already, I thought it might be useful for others.

The premise is that you want to add a tool tip to your visualization using the method described here, but you also want to include an HTML link in the tooltip that will link somewhere else. This might look a little like the following;

Tool tip with an HTML Link

[104] http://www.d3noob.org/2013/01/adding-tooltips-to-d3js-graph.html

In the image above the date has been turned into a link. In this case the link goes to google.com, but that can obviously be configurable.

The full code for this example can be found on github[105] or in the code samples bundled with this book (tooltips-link.html and data.csv). A working example can be found on bl.ocks.org[106].

There are a few changes that we would want to make to our original tooltip code to implement this feature.

First of all, we'll add the link to the date element. Adding an HTML link can be as simple as wrapping the 'thing' to be used as a link in `<a>` tags[107] with an appropriate URL to go to.

The following adaptation of the code that prints the information into our tooltip code does just that;

```
div .html(
    '<a href= "http://google.com">' + // The first <a> tag
    formatTime(d.date) +
    "</a>" +                          // closing </a> tag
    "<br/>"  + d.close)
    .style("left", (d3.event.pageX) + "px")
    .style("top", (d3.event.pageY - 28) + "px");
```

`<a href= "http://google.com">` places our first `<a>` tag and declares the URL and the second tag follows after the date.

The second change we will want to make is to ensure that the tooltip stays in place long enough for us to actually click on the link. The problem being solved here is that our original code relies on the mouse being over the dot on the graph to display the tooltip. if the tooltip is displayed and the cursor moves to press the link, it will move off the dot on the graph and the tooltip vanishes (Nice!).

To solve the problem we can leave the tooltip in place adjacent to a dot while the mouse roams freely over the graph until the next time it reaches a dot and then the previous tooltip vanishes and a new one appears. The best way to appreciate this difference is to check out the live example on bl.ocks.org[108].

The code is as follows (you may notice that this also includes the link as described above);

[105] https://gist.github.com/d3noob/c37cb8e630aaef7df30d
[106] http://bl.ocks.org/d3noob/c37cb8e630aaef7df30d
[107] http://www.w3schools.com/tags/tag_a.asp
[108] http://bl.ocks.org/d3noob/c37cb8e630aaef7df30d

```
.on("mouseover", function(d) {
    div.transition()
        .duration(500)
        .style("opacity", 0);
    div.transition()
        .duration(200)
        .style("opacity", .9);
    div .html(
        '<a href= "http://google.com">' + // The first <a> tag
        formatTime(d.date) +
        "</a>" +                          // closing </a> tag
        "<br/>"  + d.close)
        .style("left", (d3.event.pageX) + "px")
        .style("top", (d3.event.pageY - 28) + "px");
    });
```

We have removed the `.on("mouseout"` portion and moved the function that it used to carry out to the start of the `.on("mouseover"` portion. That way the first thing that occurs when the mouse cursor moves over a dot is that it removes the previous tooltip and then it places the new one.

The last change we need to make is to remove from the `<style>` section the line that told the mouse to ignore the tooltip;

```
/*  pointer-events: none;    This line needs to be removed */
```

In this case I have just commented it out so that it's a bit more obvious that it gets removed.

Moar Links!

One link is interesting, but let's face it, we didn't go to all the trouble of putting a link into a tool tip to just go to one location. Now we shift it up a gear and start linking to different places depending on our data. At the same time (and because someone asked) we will make the link open in a new tab!

The changes to the script are fairly minor, but one fairly large change is the need to have links to go to. For this example I have added a range of links to visit to our csv file so it now looks like this;

```
date,close,link
1-May-12,58.13,http://bl.ocks.org/d3noob/c37cb8e630aaef7df30d
30-Apr-12,53.98,http://bl.ocks.org/d3noob/11313583
27-Apr-12,67.00,http://bl.ocks.org/d3noob/11306153
26-Apr-12,89.70,http://bl.ocks.org/d3noob/11137963
25-Apr-12,99.00,http://bl.ocks.org/d3noob/10633856
24-Apr-12,130.28,http://bl.ocks.org/d3noob/10633704
23-Apr-12,166.70,http://bl.ocks.org/d3noob/10633421
20-Apr-12,234.98,http://bl.ocks.org/d3noob/10633192
19-Apr-12,345.44,http://bl.ocks.org/d3noob/10632804
18-Apr-12,443.34,http://bl.ocks.org/d3noob/9692795
17-Apr-12,543.70,http://bl.ocks.org/d3noob/9267535
16-Apr-12,580.13,http://bl.ocks.org/d3noob/9211665
13-Apr-12,605.23,http://bl.ocks.org/d3noob/9167301
12-Apr-12,622.77,http://bl.ocks.org/d3noob/8603837
11-Apr-12,626.20,http://bl.ocks.org/d3noob/8375092
10-Apr-12,628.44,http://bl.ocks.org/d3noob/8329447
9-Apr-12,636.23,http://bl.ocks.org/d3noob/8329404
5-Apr-12,633.68,http://bl.ocks.org/d3noob/8150631
4-Apr-12,624.31,http://bl.ocks.org/d3noob/8273682
3-Apr-12,629.32,http://bl.ocks.org/d3noob/7845954
2-Apr-12,618.63,http://bl.ocks.org/d3noob/6584483
30-Mar-12,599.55,http://bl.ocks.org/d3noob/5893649
29-Mar-12,609.86,http://bl.ocks.org/d3noob/6077996
28-Mar-12,617.62,http://bl.ocks.org/d3noob/5193723
27-Mar-12,614.48,http://bl.ocks.org/d3noob/5141528
26-Mar-12,606.98,http://bl.ocks.org/d3noob/5028304
```

The code change is to the piece of JavaScript where we add the HTML. This is what we end up with;

```
div .html(
    '<a href= "'+d.link+'" target="_blank">' + //with a link
    formatTime(d.date) +
    "</a>" +
    "<br/>"  + d.close)
    .style("left", (d3.event.pageX) + "px")
    .style("top", (d3.event.pageY - 28) + "px");
```

We've replaced the URL `http://google.com` with the variable for our `link` column `d.link` and we've also added in the `target="_blank"` statement so that our link opens in a new tab.

A quick word of warning on this piece of code. it can look a little messy because it includes nested speech-marks and quotation marks. This makes it a bit confusing if you're unused to the idea of nesting this type of information. If things aren't working in this part of your code, I recommend going back to basics and adding one piece at a time, getting it right and then add the next piece. Take it slow :-).

The full code for this multi link example can be found on github[109] or in the code samples bundled with this book (tooltips-link-multi.html and datatips.csv). A working example can be found on bl.ocks.org[110].

Hopefully that helps people with a similar desire to include links in their tooltips. Many thanks to the reader who suggested it :-).

[109] https://gist.github.com/d3noob/2e224baa7472c5e9e0e8
[110] http://bl.ocks.org/d3noob/2e224baa7472c5e9e0e8

What are the predefined, named colours?

Throughout this document I generally use colours defined by name. This is mainly because I can, and not for any other reason. In fact there several different ways to define colours used in D3 / JavaScript / CSS and HTML. I have no idea what the limitations for use are and / or how their use in different browsers impacts on correct representation. But I do know that they're used widely.

There seems to be several different standards for what constitute an authoritative list of named colours. After a cursory search I was able to find a great list on about.com[111] and there are some nice representations on Wikipedia[112].

The overriding point of all this is that there's more than one way to define colours in your graphs.

It means that considering... `.style("fill", "steelblue")`
and...
`.style("fill", "#4682b4")`
and...
`.style("fill", "rgb(70,130,180)")`

All three alternatives result in the same colour being applied.

For a long time I didn't actually have the images of the colours represented here in D3 Tips and Tricks, but like all things, one day I thought 'Hey, I could just write a simple script that placed them on the screen'. So here they are :-).

I have tried to group them as 'like' colours per the entry in Wikipedia.

[111] http://webdesign.about.com/od/colorcharts/l/bl_namedcolors.htm
[112] http://en.wikipedia.org/wiki/Web_colors#X11_color_names

LightYellow	LemonChiffon	LightGoldenrodYellow
PapayaWhip	Moccasin	PeachPuff
PaleGoldenrod	Khaki	DarkKhaki
Gold	Cornsilk	BlanchedAlmond
Bisque	NavajoWhite	Wheat
BurlyWood	Tan	RosyBrown
SandyBrown	Goldenrod	DarkGoldenrod
Peru	Chocolate	SaddleBrown
Sienna	Brown	Maroon
DarkOliveGreen	Olive	OliveDrab
YellowGreen	LimeGreen	Lime
LawnGreen	Chartreuse	GreenYellow
SpringGreen	MediumSpringGreen	LightGreen
PaleGreen	DarkSeaGreen	MediumSeaGreen
SeaGreen	ForestGreen	Green
DarkGreen	MediumAquamarine	Aqua

Cyan	LightCyan	PaleTurquoise
Aquamarine	Turquoise	MediumTurquoise
DarkTurquoise	LightSeaGreen	CadetBlue
DarkCyan	Teal	LightSteelBlue
PowderBlue	LightBlue	SkyBlue
LightSkyBlue	DeepSkyBlue	DodgerBlue
CornflowerBlue	SteelBlue	RoyalBlue
Blue	MediumBlue	DarkBlue
Navy	MidnightBlue	Lavender
Thistle	Plum	Violet
Orchid	Fuchsia	Magenta
MediumOrchid	MediumPurple	BlueViolet
DarkViolet	DarkOrchid	DarkMagenta
Purple	Indigo	DarkSlateBlue
Snow	Honeydew	MintCream
Azure	AliceBlue	GhostWhite

You can also see a live page with the script that produces the rectangles at bl.ocks.org[113].

[113] http://bl.ocks.org/d3noob/11313583

Selecting / filtering a subset of objects

OK, Imagine a scenario where you want to select (or should we say filter) a particular range of objects from a larger set.

For example, what if we wanted to use our scatter plot example to show the line as normal, but we are particularly interested in the points where the values of the points fall below 400. And when it does we want them highlighted with a circle as we have done with *all* the points previously.

So that we end up with something that looks a little like this…

Only the points below 400 are selected

Err… Yes, for those among you who are of the observant persuasion, I have deliberately coloured them red as well (red for **DANGER**!).

This is a fairly simple example, but serves to illustrate the principle adequately. From our simple scatter plot example we only need to add in two lines to the block of code that draws the circles as follows;

```
svg.selectAll("dot")
    .data(data)
.enter().append("circle")
.filter(function(d) { return d.close < 400 })    // <== This line
    .style("fill", "red")                        // <== and this one
    .attr("r", 3.5)
    .attr("cx", function(d) { return x(d.date); })
    .attr("cy", function(d) { return y(d.close); });
```

The full code for this example can be found on github[114] or in the code samples bundled with this book (filter-selection.html and data.csv). A working example can be found on bl.ocks.org[115].

The first added line uses the `.filter` function to act on the data points and according to the arguments passed to it in this case, only return those where the value of d.close is less than 400 (`return d.close < 400`).

[114] https://gist.github.com/d3noob/8dc93bce7e7200ab487d
[115] http://bl.ocks.org/d3noob/8dc93bce7e7200ab487d

The second added line is our line that simply colours the circles red (`.style("fill", "red")`).

That's all there is to it. Pretty simple, but the filter function can be very powerful when used wisely.

I've placed a copy of the file for selecting / filtering into the downloads section on d3noob.org with the general examples as filter-selection.html.

Select items with an IF statement.

The filtering – selection section above is a good way to adapt what you see on a graph, but so is a more familiar friend… The 'if' statement.

An if statement will act to carry out a task in a particular way dependant on a condition that you specify.

Here's an example, what if we wanted to show our scatter plot as normal, but all those with a 'close' value less than 400 should be coloured red. Sound familiar? Yes, I know it's similar to the example above, with the subtle difference that it is leaving the circles above 400 in place (more on that to follow).

Starting with the simple scatter plot example all we have to do is include the if statement in the block of code that draws the circles. Here's the entire block with the additions highlighted;

```
svg.selectAll("dot")
    .data(data)
.enter().append("circle")
    .attr("r", 3.5)
    .style("fill", function(d) {            // <== Add these
        if (d.close <= 400) {return "red"}  // <== Add these
        else         { return "black" }          // <== Add these
    ;})                                     // <== Add these
    .attr("cx", function(d) { return x(d.date); })
    .attr("cy", function(d) { return y(d.close); });
```

Our first added line introduces the style modifier and the rest of the code acts to provide a return for the 'fill' attribute.

The second line introduces our if statement. There's very little difference using `if` statements between languages. Just look out for maintaining the correct syntax and you should be fine. In this case we're asking if the value of d.close is less than or equal to 400 and if it is it will return the `"red"` statement for our fill.

The third line covers our rear and make sure that if the colour isn't going to be red, it's going to be black. The last line just closes the style and function statements.

The result?

Points above 400 black and points below 400 red

Aww..... nice.

The full code for this example can be found on github[116] or in the code samples bundled with this book (if-selection.html and data.csv). A working example can be found on bl.ocks.org[117].

Could it be any cooler? I'm glad you asked.

What if we wanted to have all the points where close was less than 400 red and all those where close was greater than 620 green? Oh yeah! Now we're talking.

So with one small change to the if statement;

```
.style("fill", function(d) {
    if (d.close <= 400) {return "red"}
    else if (d.close >= 620) {return "lawngreen"} // <== Right here
    else { return "black" }
;})
```

Check it out...

Points coloured differently depending on their value

Nice.

[116] https://gist.github.com/d3noob/464c92429ac05c6a19a1
[117] http://bl.ocks.org/d3noob/464c92429ac05c6a19a1

Applying a colour gradient to a line based on value.

I just know that you were impressed with the changing dots in a scatter plot based on the value. But could we go one better?

How about we try to reproduce the same effect but by varying the colour of the plotted line. This is a neat feature and a useful example of the flexibility of d3.js and SVG in general. I used the appropriate bits of code from Mike Bostock's Threshold Encoding example[118]. And I should take the opportunity to heartily recommend browsing through his collection of examples on bl.ocks.org[119]. For those who prefer to see the code in it's fullest, there is an example as an appendix (Graph with Area Gradient) that can assist (although it is for a later example that uses a gradient in a similar way (don't worry we'll get to it in a few pages)).

Here then is a plotted line that is red below 400, green above 620 and black in between.

Line colour varied with gradient

How cool is that?

Enough beating around the bush, how is the magic line produced?

Starting with our simple line graph, there are only two blocks of code to go in. One is CSS in the `<style>` area and the second is a tricky little piece of code that deals with gradients.

The full code for this example can be found on github[120] or in the code samples bundled with this book (line-colour-gradient-graph.html and data.csv). A working example can be found on bl.ocks.org[121].

So, first the CSS.

```
.line {
    fill: none;
    stroke: url(#line-gradient);
    stroke-width: 2px;
}
```

[118] http://bl.ocks.org/3970883
[119] http://bl.ocks.org/mbostock
[120] https://gist.github.com/d3noob/3e72cafd95e1834f599b
[121] http://bl.ocks.org/d3noob/3e72cafd95e1834f599b

This block can go in the `<style>` area towards the end.

There's the fairly standard fill of none and a stroke width of `2` pixels, but the `stroke: url(#line-gradient);` is something different.

In this case the stroke (the colour of the line) is being determined at a link within the page which is set by the anchor `#line-gradient`. We will see shortly that this is in our second block of code, so the colour is being defined in a separate portion of the script.

And now the JavaScript gradient code;

```
svg.append("linearGradient")
    .attr("id", "line-gradient")
    .attr("gradientUnits", "userSpaceOnUse")
    .attr("x1", 0).attr("y1", y(0))
    .attr("x2", 0).attr("y2", y(1000))
.selectAll("stop")
    .data([
        {offset: "0%", color: "red"},
        {offset: "40%", color: "red"},
        {offset: "40%", color: "black"},
        {offset: "62%", color: "black"},
        {offset: "62%", color: "lawngreen"},
        {offset: "100%", color: "lawngreen"}
    ])
.enter().append("stop")
    .attr("offset", function(d) { return d.offset; })
    .attr("stop-color", function(d) { return d.color; });
```

There's our anchor on the second line!

But let's not get ahead of ourselves. This block should be placed after the x and y domains are set, but before the line is drawn.

Seems a bit strange doesn't it? This block is all about defining the actions of an element, but the element in this case is a gradient and the gradient acts on the line.

So, our first line adds our linear gradient. Gradients consist of continuously smooth colour transitions along a vector from one colour to another We can have a linear one or a radial one and depending on which you select, there are a few options to define. There is some great information on gradients at http://www.w3.org/TR/SVG/pservers.html[122] (more than I ever thought existed).

The second line (`.attr("id", "line-gradient")`) sets our anchor for the CSS that we saw earlier.

The third fourth and fifth lines define the bounds of the area over which the gradient will act. Since the coordinates `x1`, `y1`, `x2`, `y2` will describe an area. The values for `y1` (0) and `y2` (1000) are used more for convenience to align with our data (which has a maximum value around 630 or so). For more information on the gradientUnits attribute I found this page useful https://developer.mozilla.org/en-US/docs/SVG/Attribute/gradientUnits[123]. We'll come back to

[122] http://www.w3.org/TR/SVG/pservers.html
[123] https://developer.mozilla.org/en-US/docs/SVG/Attribute/gradientUnits

the coordinates in a moment.

The next block selects all the 'stop' elements for the gradients. These stop elements define where on the range covered by our coordinates the colours start and stop. These have to be defined as either percentages or numbers (where the numbers are really just percentages in disguise (i.e. 45% =0.45)).

The best way to consider the stop elements is in conjunction with the `gradientUnits`. The image following may help.

Varying colours for varying values make a gradient

In this case our coordinates describe a vertical line from 0 to 1000. Our colours transition from red (0) to red (400) at which point they change to black (400) and this will continue until it gets to black (620). Then this changes to green (620) and from there, any value above that will be green.

Now, it might seem a little convoluted to be doubling up on the colours and values, but the reason is that the gradient functions have a lot more to them than we're using and we'll have a look at the possibilities once the explanation of the code is done.

So after defining the stop elements, we enter and append the elements to the gradient (`.enter().append("stop")`) with attributes for offset and colour that we defined in the stop elements area.

Now, that *IS* cool, but by now, I hope that you have picked that a gradient function really does mean a gradient, and not just a straight change from one colour to another.

So, let's try changing the stop element offsets to the following (and making the stroke-width slightly larger to see more clearly what's going on);

```
.data([
    {offset: "0%", color: "red"},
    {offset: "30%", color: "red"},
    {offset: "45%", color: "black"},
    {offset: "55%", color: "black"},
    {offset: "60%", color: "lawngreen"},
    {offset: "100%", color: "lawngreen"}
])
```

And here we go...

Line with a gradually changing gradient

Ahh... A real gradient.

I have tended to find that I need to have a good think about how I set the offsets and bounds when doing this sort of thing since it can get quite complicated quite quickly :-)

Applying a colour gradient to an area fill.

The previous example of a varying gradient on a line is neat, but hopefully you're already thinking "Hang on, can't that same thing be applied to an area fill?".

Damn! You're catching on.

To do this there's only a few things we need to change;

First of all the CSS for the line needs to be amended to refer to the area. So this...

```
.line {
    fill: none;
    stroke: url(#line-gradient);
    stroke-width: 2px;
}
```

...gets changed to this...

```
.area {
    fill: url(#area-gradient);
    stroke-width: 0px;
}
```

We've defined the styles for the area this time, but instead of the stroke being defined by the separate script, now it's the area. While we've changed the url name, it's actually the same piece of code, with a different id (because it seemed wrong to be talking about an area when the label said line). We've also set the stroke width to zero, because we don't want any lines around our filled area.

Now we want to take the block of code that defined our line...

```
var        valueline = d3.svg.line()
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y(d.close); });
```

... and we need to replace it with the standard block that defined an area fill.

```
var        area = d3.svg.area()
    .x(function(d) { return x(d.date); })
    .y0(height)
    .y1(function(d) { return y(d.close); });
```

So we're not going to be drawing a line at all. Just the area fill.

Next, as I mentioned earlier, we change the id for the `linearGradient` block from `"line-gradient"` to `"area-gradient"`

```
        .attr("id", "area-gradient")
```

And lastly, we remove the block of code that drew the line and replace it with a block that draws an area. So change this....

```
svg.append("path")
    .attr("class", "line")
    .attr("d", valueline(data));
```

... to this;

```
svg.append("path")
    .datum(data)
    .attr("class", "area")
    .attr("d", area);
```

And then sit back and marvel at your creation;

Area fill with a gradually changing gradient

The full code for this example can be found on github[124] or in the code samples bundled with this book (area-colour-gradient-graph.html and data.csv). A working example can be found on bl.ocks.org[125].

For a slightly 'nicer' looking example, you could check out a variation of one of Mike Bostocks originals here; http://bl.ocks.org/4433087[126].

[124] https://gist.github.com/d3noob/9fcf79b76c2643b98dd7
[125] http://bl.ocks.org/d3noob/9fcf79b76c2643b98dd7
[126] http://bl.ocks.org/4433087

Show / hide an element by clicking on another element

This is a trick that I found I wanted to impliment in order to present a graph with a range of lines and to then provide the reader with the facility to click on the associated legend to toggle the visibility of the lines off and on as required.

The example we'll follow is our friend from earlier, a slightly modified example of the graph with two lines.

Show / hide lines on a graph

In this example we will be able to click on either of the two titles at the bottom of the graph ('Blue Line' or 'Red Line') and have it toggle the respective line and Y axis.

The code

The code for the example is available online at bl.ocks.org[127] or GitHub[128]. It is also available as the file 'show-hide.html' as a separate download with D3 Tips and Tricks. A copy of most the files that appear in the book can be downloaded (in a zip file) when you download the book from Leanpub[129].

There are two main parts to implementing this technique. Firstly we have to label the element (or elements) that we wish to show / hide and then we have to give the object that will get clicked on the attribute that allows it to recognise a mouse click and the code that it subsequently uses to show / hide our labelled element.

Labelling the element that is to be switched on and off is dreadfully easy. It simply involves including an `id` attribute to an element that identifies it uniquely.

```
svg.append("path")
    .attr("class", "line")
    .attr("id", "blueLine")
    .attr("d", valueline(data));
```

In the example above we have applied the `id blueLine` to the path that draws the blue line on our graph.

The second part is a little trickier. The following is the portion of JavaScript that places our text label under the graph. The only part of it that is unusual is the `.on("click", function()` section of the code.

[127] http://bl.ocks.org/d3noob/5d621a60e2d1d02086bf
[128] https://gist.github.com/d3noob/5d621a60e2d1d02086bf
[129] https://leanpub.com/D3-Tips-and-Tricks

```
svg.append("text")
    .attr("x", 0)
    .attr("y", height + margin.top + 10)
    .attr("class", "legend")
    .style("fill", "steelblue")
    .on("click", function(){
        // Determine if current line is visible
        var active   = blueLine.active ? false : true,
          newOpacity = active ? 0 : 1;
        // Hide or show the elements
        d3.select("#blueLine").style("opacity", newOpacity);
        d3.select("#blueAxis").style("opacity", newOpacity);
        // Update whether or not the elements are active
        blueLine.active = active;
```

When we click on our 'Blue Line' text element the `.on("click", function()` section executes.

We're using a short-hand version of the `if` statement a couple of times here. Firstly we check to see if the variable `blueLine.active` is true or false and if it's true it gets set to false and if it's false it gets set to true (not at all confusing).

```
        var active   = blueLine.active ? false : true,
          newOpacity = active ? 0 : 1;
```

Then after toggling this variable we set the value of `newOpacity` to either `0` or `1` depending on whether `active` is `false` or `true` (the second short-hand JavaScript if statement).

We can then select our identifiers that we have declared using the `id` attributes in the earlier pieces of code and modify their opacity to either `0` (off) or `1` (on)

```
        d3.select("#blueLine").style("opacity", newOpacity);
        d3.select("#blueAxis").style("opacity", newOpacity);
```

Lastly we update our `blueLine.active` variable to whatever the `active` state is so that it can toggle correctly the next time it is clicked on.

```
        blueLine.active = active;
```

Quite a neat piece of code. Kudos to Max Leiserson for providing the example on which it is largely based in an answer to a question on Stack Overflow[130].

[130] http://stackoverflow.com/questions/20249215/how-to-display-and-hide-links-and-nodes-when-clicking-on-a-node-in-d3-javascript

Export an image from a d3.js page as a SVG or bitmap

At some point you will want to take your lovingly crafted D3 graphical masterpiece and put it in a (close your eyes if you're squeamish) Power Point presentation or Word document or export it for sharing in some other way.

There could be many reasons for wanting to do this and some may be more complicated than I will be willing to explore, but for the occasional conversion of images I have found what I regard as a fairly easy process.

Before we begin our exporting odyssey, let's cover a little bit of housekeeping and describe the difference between a vector graphic (in this case specifically Scalable Vector Graphics) and a bitmap. Please skip ahead if you're comfortable with the terms.

Bitmaps

A bitmap (or raster) image is one that is composed of lots of discrete individual dots (let's call them pixels) which, when joined together (and zoomed out a bit) give the impression of an image. If we use the example of the force layout example we developed, and look at a screen shot (and it's important to remember that this is a screen shot) of the image we see a picture that looks fairly benign.

A bitmap at a normal zoom level

However, as we enlarge the image by doubling it's size (x 2) we begin to see some rough edges appear.

A bitmap at 200%

And if we enlarge it by doubling again (x 4) , it starts to look decidedly rough.

A bitmap at 400%

Doubling again (x 8), starts to show the pixels pretty clearly.

A bitmap at 800%

Doubling again for the last time (x 16) and the pixels are plainly evident.

A bitmap at 1600%

Bitmaps can be saved in a wide range of formats depending on users requirements including compression, colour depth, transparency and a host of other attributes. Typically they can be identified by the file suffix .jpg, .png or .bmp (and there are an equally large number of other suffixes).

This will be the type of format that most people will be familiar with for images and their ubiquity with the advent of digital cameras almost makes it redundant to describe them.

However, there is another type of image and it is even more important to d3.js users.

Vector Graphics (Specifically SVG)

Scalable Vector Graphics (SVG) use a technique of drawing an image that relies more on a description of an image than the final representation that a user sees. Instead of arranging individual pixels, an image is created by describing the way the image is created.

For instance, drawing a line would be accomplished by defining two sets of coordinates and specifying a line of a particular width and colour be drawn between the points.

This might sound a little long winded, and it does create a sense of abstraction, but it is a far more powerful mechanism for drawing as there is no loss of detail with increasing scale. Changes to the image can be simply carried out by adjusting the coordinates, colour description, line width or curve diameter. If this all sounds a little familiar, you have *definitely* been paying attention, because this is the heart of the way that d3.js draws images in a browser. It uses a combination of coordinates, shapes and attributes to create vector images in a web page.

As a demonstration of the difference, here is the same original picture which I have saved as a SVG image.

A SVG image at a normal zoom level

Enlarged by doubling it's size (x 2) everything looks smooth.

A SVG at 200%

If we enlarge it by doubling again (x 4) , it still looks good.

A SVG at 400%

Doubling again (x 8) and we can see that the text 'James' is actually composed of a fill colour and a border.

A SVG at 800%

Doubling again for the last time (x 16) everything still retains it's clear sharp edges.

A SVG at 1600%

Let's get exporting!

We'll use a three stage process for exporting our image (assuming the desired end result is a bitmap) and usefully, the first stage will result in us having a vector image as well!

The sequence will go as follows:

1. Copy the image from the web page and save it as a SVG file

2. Open the SVG image in a program designed to use vector images and edit it if required.
3. Export that image as a bitmap

Copying the image off the web page

Getting the image out of a web page is made easy by using 'SVG Crowbar[131]'. This is a "*A Chrome-specific bookmarklet that extracts SVG nodes and accompanying styles from an HTML document and downloads them as an SVG file*". What that means is that once you drag the bookmarklet from the web page to your bookmarks (You need to be using Google Chrome, and I'm told that about 60% of the people who visit d3noob.org do) you're ready to go.

Drag the 'SVG Crowbar' Object from the web page to your bookmarks bar

Now when you have a web page open that's displaying a D3 creation, all you need to do is click on the SVG Crowbar bookmark and you will be prompted for a location to save a svg image.

Really. It's that simple.

Open the SVG Image and Edit

Obviously now that you have a SVG image, you need to be able to do something with it. My preferred software for this is Inkscape[132].

Inkscape is "*An Open Source vector graphics editor, with capabilities similar to Illustrator, CorelDraw, or Xara X, using the W3C standard Scalable Vector Graphics (SVG) file format*".

It really is an extremely capable drawing program and it is capable of a lot more than the job we're going to use it for, so you may find it has other uses that may be valuable.

Once installed, you can open the saved file directly into Inkscape.

[131] http://nytimes.github.io/svg-crowbar/
[132] http://inkscape.org/

Inkscape with our force diagram

While here you can edit the drawing to your hearts delight. I particularly recommend ungrouping the diagram and removing or adjusting individual elements if required.

Once you have finished editing, you are ready for the final step.

Saving as a bitmap

While still in Inkscape, go to the 'File', 'Export Bitmap...' menu.

Inkscape Export Bitmap menu

This will open a dialog box where you can select an appropriate resolution and location for your bitmap and then press the export button.

Inkscape Export Bitmap dialog

There you go.

It is worth knowing that the default settings here will export the diagram with a transparent background (using *.png) which will fit in nicely with a wide range of graphical end uses.

Using HTML inputs with d3.js

Part of the attraction of using technologies like d3.js is that it expands the scope of what is possible in a web page. At the same time, there are many different options for displaying content on a page and plenty of ways of interacting with it.

Some of the most basic of capabilities has been the use of HTML entities that allow the entry of data on a page. This can take a range of different forms (pun intended) and the `<input>` tag is one of the most basic.

What is an HTML input?

An HTML input is an element in HTML that allows a web page to input data. There are a range of different input types (with varying degrees of compatibility with browsers) and they are typically utilised inside a `<form>` element.

For example the following code allows a web page to place two fields on a web page so that a user can enter their first and last names in separate boxes;

```
<form>
  First name: <input type="text" name="firstname"><br>
  Last name: <input type="text" name="lastname">
</form>
```

The page would then display the following;

A form input

The range of input types is large and includes;

- text: A simple text field that a user can enter information into.
- radio: Buttons that let a user select only one of a limited number of choices.
- button: A clickable button that can activate JavaScript.
- range: A slider control for setting a number whose exact value is not important.
- number: A field for entering a number or toggling a number up and down.

... and many more. To check out others and get further background, it would be worth while visiting theMozilla developer[133] pages or w3schools.com[134].

While d3.js has the power to control and manipulate a web page to an extreme extent, sometimes it's desirable to use a simple process to get a result. The following explanations will demonstrate a simple use case linking an HTML input with a d3.js element and will go on to provide examples of

[133] https://developer.mozilla.org/en-US/docs/Web/HTML/Element/Input
[134] http://www.w3schools.com/tags/tag_input.asp

using multiple inputs, affecting multiple elements and using different input types. The examples are deliberately kept simple. They are intended to demonstrate functionality and to provide a starting position for you to go forward :-).

Using a `range` input with d3.js

The first example we will follow will use a `range` input to adjust the radius of a circle.

Adjust the radius of a circle

The code

The following is the full code for the example. A live version is available online at bl.ocks.org[135] or GitHub[136]. It is also available as the file 'input-radius.html' as a separate download with D3 Tips and Tricks. A copy of most the files that appear in the book can be downloaded (in a zip file) when you download the book from Leanpub[137].

```
<!DOCTYPE html>
<meta charset="utf-8">
<title>Input test (circle)</title>

<p>
  <label for="nRadius"
         style="display: inline-block; width: 240px; text-align: right">
         radius = <span id="nRadius-value">…</span>
  </label>
  <input type="range" min="1" max="150" id="nRadius">
</p>

<script src="http://d3js.org/d3.v3.min.js"></script>
<script>
```

[135] http://bl.ocks.org/d3noob/10632804
[136] https://gist.github.com/d3noob/10632804
[137] https://leanpub.com/D3-Tips-and-Tricks

```
var width = 600;
var height = 300;

var holder = d3.select("body")
      .append("svg")
      .attr("width", width)
      .attr("height", height);

// draw the circle
holder.append("circle")
  .attr("cx", 300)
  .attr("cy", 150)
  .style("fill", "none")
  .style("stroke", "blue")
  .attr("r", 120);

// when the input range changes update the circle
d3.select("#nRadius").on("input", function() {
  update(+this.value);
});

// Initial starting radius of the circle
update(120);

// update the elements
function update(nRadius) {

  // adjust the text on the range slider
  d3.select("#nRadius-value").text(nRadius);
  d3.select("#nRadius").property("value", nRadius);

  // update the circle radius
  holder.selectAll("circle")
    .attr("r", nRadius);
}

</script>
```

The explanation

As with the other examples in the book I will not go over some of the simpler lines of code that are covered in greater detail in earlier sections of the book and will concentrate on those sections that contain new concepts, code or look like they might need expanding :-).

The first section is the portion that sets out the html `range` input;

```
<p>
  <label for="nRadius"
         style="display: inline-block; width: 240px; text-align: right">
         radius = <span id="nRadius-value">…</span>
  </label>
  <input type="range" min="1" max="150" id="nRadius">
</p>
```

The entire block is enclosed in a paragraph (`<p>`) tag so that is appears on a single line. It can be broken down into the label that occurs before the input slider which is given the id `nRadius-value` and the `input` proper.

The `for` attribute of the `label` tag equals to the id attribute of the input element to bind them together. This allows us to update the text later as the slider is moved.

The `input` tag can include four attributes that specify restrictions on the operation of the slider;

- `max`: specifies the maximum value allowed
- `min`: specifies the minimum value allowed
- `step`: specifies the number intervals as you move the slider
- `value`: Specifies the default value

The `ids` supplied for both the label and the input are important since they provide the reference for our d3.js script.

The first portion of our JavaScript is fairly routine if you've been following along with the rest of the book.

```
var width = 600;
var height = 300;

var holder = d3.select("body")
      .append("svg")
      .attr("width", width)
      .attr("height", height);

// draw the circle
holder.append("circle")
  .attr("cx", 300)
  .attr("cy", 150)
  .style("fill", "none")
  .style("stroke", "blue")
  .attr("r", 120);
```

We append an SVG element to the `body` of our page and then we append a circle with some particular styling to the SVG element.

Then things start to get more interesting…

```
d3.select("#nRadius").on("input", function() {
  update(+this.value);
});
```

We select our input using the `id` that we had declared earlier in the html (`nRadius`). Then we use the `.on` operator which adds what is called an 'event listener[138]' to the element so that when there is a change in the element (in this case an adjustment of the slider of the input) a function is called (`function()`) that in turn calls the `update` function with the value from the input (`+this.value`). We haven't seen the `update` function yet, but never fear, it's coming.

We also call the update function with a specific value in the next line;

```
update(120);
```

This might seem slightly redundant, but unless the function gets a value, the text associated with the range input doesn't get a reading and remains on '...' until the slider is moved.

Lastly we have our `update` function;

```
function update(nRadius) {

  // adjust the text on the range slider
  d3.select("#nRadius-value").text(nRadius);
  d3.select("#nRadius").property("value", nRadius);

  // update the circle radius
  holder.selectAll("circle")
    .attr("r", nRadius);
}
```

The first part of the function selects the `label` associated with our input (with the `id`, `nRadius-value`) and applies the vaule that has been passed into the function (`nRadius`). The next line selects the input itself and applies the value to it (this would be the equivalent of having `value="<number here>"` as a property in the html).

Lastly, we select the circle element and apply the new radius value based on our input value `nRadius` (`.attr("r", nRadius)`).

And there we have it, a fully adjustable radius for our circle controlled with an HTML input.

[138] https://github.com/mbostock/d3/wiki/Selections#on

Maximum radius for our circle

Using more than one input

In this example we will use two separate inputs (range type) to adjust the height and width of a rectangle.

Dual inputs

This is not too much of a stretch from the previous single input example with the radius of a circle, but it may be useful to reinforce the concept and illustrate something slightly different.

The code

The following is the full code for the example. A live version is available online at bl.ocks.org[139] or GitHub[140]. It is also available as the file 'input-double.html' as a separate download with D3 Tips and Tricks. A copy of most the files that appear in the book can be downloaded (in a zip file) when you download the book from Leanpub[141].

```
<!DOCTYPE html>
<meta charset="utf-8">
<title>Double Input Test</title>

<p>
  <label for="nHeight"
         style="display: inline-block; width: 240px; text-align: right">
         height = <span id="nHeight-value">…</span>
  </label>
  <input type="range" min="1" max="280" id="nHeight">
</p>
```

[139] http://bl.ocks.org/d3noob/10633192
[140] https://gist.github.com/d3noob/10633192
[141] https://leanpub.com/D3-Tips-and-Tricks

```
<p>
  <label for="nWidth"
         style="display: inline-block; width: 240px; text-align: right">
         width = <span id="nWidth-value">…</span>
  </label>
  <input type="range" min="1" max="400" id="nWidth">
</p>

<script src="http://d3js.org/d3.v3.min.js"></script>
<script>

var width = 600;
var height = 300;

var holder = d3.select("body")
      .append("svg")
      .attr("width", width)
      .attr("height", height);

// draw a rectangle
holder.append("rect")
    .attr("x", 300)
    .attr("y", 150)
    .style("fill", "none")
    .style("stroke", "blue")
    .attr("height", 150)
    .attr("width", 200);

// read a change in the height input
d3.select("#nHeight").on("input", function() {
  updateHeight(+this.value);
});

// read a change in the width input
d3.select("#nWidth").on("input", function() {
  updateWidth(+this.value);
});

// update the values
updateHeight(150);
updateWidth(100);

// Update the height attributes
function updateHeight(nHeight) {

  // adjust the text on the range slider
```

```
  d3.select("#nHeight-value").text(nHeight);
  d3.select("#nHeight").property("value", nHeight);

  // update the rectangle height
  holder.selectAll("rect")
    .attr("y", 150-(nHeight/2))
    .attr("height", nHeight);
}

// Update the width attributes
function updateWidth(nWidth) {

  // adjust the text on the range slider
  d3.select("#nWidth-value").text(nWidth);
  d3.select("#nWidth").property("value", nWidth);

  // update the rectangle width
  holder.selectAll("rect")
    .attr("x", 300-(nWidth/2))
    .attr("width", nWidth);
}

</script>
```

The explanation

For the sake of brevity, this explanation will simply concentrate on the differences between the previous single input example and this one.

The declarations for the inputs in the HTML at the start of the code are simply duplicates of each other in terms of function;

```
<p>
  <label for="nHeight"
         style="display: inline-block; width: 240px; text-align: right">
         height = <span id="nHeight-value">…</span>
  </label>
  <input type="range" min="1" max="280" id="nHeight">
</p>

<p>
  <label for="nWidth"
         style="display: inline-block; width: 240px; text-align: right">
         width = <span id="nWidth-value">…</span>
  </label>
  <input type="range" min="1" max="400" id="nWidth">
</p>
```

The only significant difference is the declaration of the `id`'s for each input and it's respective label.

The JavaScript selection of the inputs is more duplication;

```
d3.select("#nHeight").on("input", function() {
  updateHeight(+this.value);
});

d3.select("#nWidth").on("input", function() {
  updateWidth(+this.value);
});
```

Again the only substantive difference is the use of the appropriate `id` values.

The updating of the width and height is done via two different functions;

```
function updateHeight(nHeight) {

  // adjust the text on the range slider
  d3.select("#nHeight-value").text(nHeight);
  d3.select("#nHeight").property("value", nHeight);

  // update the rectangle height
  holder.selectAll("rect")
    .attr("y", 150-(nHeight/2))
    .attr("height", nHeight);
}

// Update the width attributes
function updateWidth(nWidth) {

  // adjust the text on the range slider
  d3.select("#nWidth-value").text(nWidth);
  d3.select("#nWidth").property("value", nWidth);

  // update the rectangle width
  holder.selectAll("rect")
    .attr("x", 300-(nWidth/2))
    .attr("width", nWidth);
}
```

The rectangle is selected using a common `rect` designator, so multiple rectangles could be controlled. But each function controls only a specific attribute (height or width).

Rotate text with an input

This example is really just a derivative of the adjustment of a single attribute of an element.

I happen to think it's just a little bit 'neater' because it includes text, but in reality, it's just another attribute that can be adjusted.

Here we let our range input adjust the rotation of a piece of text.

Text rotation with an input

The explanation

We'll dispense with the full code listing since it's just a regurgitation of the adjusting of the radius of the circle example, but the code for the example is available online at bl.ocks.org[142] or GitHub[143]. It is also available as the file 'input-text-rotate.html' as a separate download with D3 Tips and Tricks. A copy of most the files that appear in the book can be downloaded (in a zip file) when you download the book from Leanpub[144].

The only, thing of even a slight difference (other than some naming conventions) is the initial drawing of the text...

```
holder.append("text")
  .style("fill", "black")
  .style("font-size", "56px")
  .attr("dy", ".35em")
  .attr("text-anchor", "middle")
  .attr("transform", "translate(300,150) rotate(0)")
  .text("d3noob.org");
```

... and the update function;

[142] http://bl.ocks.org/d3noob/10633421
[143] https://gist.github.com/d3noob/10633421
[144] https://leanpub.com/D3-Tips-and-Tricks

```
function update(nAngle) {

  // adjust the text on the range slider
  d3.select("#nAngle-value").text(nAngle);
  d3.select("#nAngle").property("value", nAngle);

  // rotate the text
  holder.select("text")
    .attr("transform", "translate(300,150) rotate("+nAngle+")");
}
```

Use a `number` input with d3.js

There are obviously different inputs that can be selected. The following example still rotates our text, but uses a `number` type of input to do it;

```
<p>
  <label for="nValue"
         style="display: inline-block; width: 240px; text-align: right">
         angle = <span id="nValue-value"></span>
  </label>
  <input type="number" min="0" max="360" step="5" value="0" id="nValue">
</p>
```

we have set the `step` value to speed things up a bit when rotating, but it's completely optional.

The input itself can be adjusted up or down using a mouse click or have a number typed into the input box.

Text rotation with a number input

This type of input is slightly different from the range type since it isn't fully supported under Firefox and as a result when I was testing it the arrow keys for going up and down weren't present.

The full code for the example is available online at bl.ocks.org[145] or GitHub[146]. It is also available as the file 'input-number-text.html' as a separate download with D3 Tips and Tricks. A copy of most the files that appear in the book can be downloaded (in a zip file) when you download the book from Leanpub[147].

[145] http://bl.ocks.org/d3noob/10633704
[146] https://gist.github.com/d3noob/10633704
[147] https://leanpub.com/D3-Tips-and-Tricks

Change more than one element with an input

The final example looking at using HTML inputs with d3.js incorporates a single input acting or two different elements. This might seem self evident, but if you're as unfamiliar with HTML as I am (it's embarrassing I know, but what can you do?) it may be of assistance.

The end result is to produce a single slider as a `range` input that rotates two separate text objects in different directions simultaneously.

Dual text rotation

The code

The following is the full code for the example. A live version is available online at bl.ocks.org[148] or GitHub[149]. It is also available as the file 'input-text-rotate-2.html' as a separate download with D3 Tips and Tricks. A copy of most the files that appear in the book can be downloaded (in a zip file) when you download the book from Leanpub[150].

```
<!DOCTYPE html>
<meta charset="utf-8">
<title>Input test</title>

<p>
  <label for="nAngle"
         style="display: inline-block; width: 240px; text-align: right">
         angle = <span id="nAngle-value">…</span>
  </label>
  <input type="range" min="0" max="360" id="nAngle">
</p>

<script src="http://d3js.org/d3.v3.min.js"></script>
<script>
```

[148] http://bl.ocks.org/d3noob/10633856
[149] https://gist.github.com/d3noob/10633856
[150] https://leanpub.com/D3-Tips-and-Tricks

```
var width = 600;
var height = 300;

var holder = d3.select("body")
      .append("svg")
      .attr("width", width)
      .attr("height", height);

// draw d3.js text
holder.append("text")
  .attr("class", "d3js")
  .style("fill", "black")
  .style("font-size", "56px")
  .attr("dy", ".35em")
  .attr("text-anchor", "middle")
  .attr("transform", "translate(300,55) rotate(0)")
  .text("d3.js");

// draw d3noob.org text
holder.append("text")
  .attr("class", "d3noob")
  .style("fill", "black")
  .style("font-size", "56px")
  .attr("dy", ".35em")
  .attr("text-anchor", "middle")
  .attr("transform", "translate(300,130) rotate(0)")
  .text("d3noob.org");

// when the input range changes update the rectangle
d3.select("#nAngle").on("input", function() {
  update(+this.value);
});

// Initial starting height of the rectangle
update(0);

// update the elements
function update(nAngle) {

// adjust the range text
  d3.select("#nAngle-value").text(nAngle);
  d3.select("#nAngle").property("value", nAngle);

  // adjust d3.js text
  holder.select("text.d3js")
```

```
    .attr("transform", "translate(300,55) rotate("+nAngle+")");

  // adjust d3noob.org text
  holder.select("text.d3noob")
    .attr("transform", "translate(300,130) rotate("+(360 - nAngle)+")");
}

</script>
```

The explanation

The explanation for this example differes from the others in the way that the d3.js elements (the two pieces of text) are initially appended and then updated.

When they are initially drawn...

```
holder.append("text")
  .attr("class", "d3js")
  .style("fill", "black")
  .style("font-size", "56px")
  .attr("dy", ".35em")
  .attr("text-anchor", "middle")
  .attr("transform", "translate(300,55) rotate(0)")
  .text("d3.js");

holder.append("text")
  .attr("class", "d3noob")
  .style("fill", "black")
  .style("font-size", "56px")
  .attr("dy", ".35em")
  .attr("text-anchor", "middle")
  .attr("transform", "translate(300,130) rotate(0)")
  .text("d3noob.org");
```

... both elements are declared with a `class` attribute that serves as a reference for the future updating. Here, the text 'd3.js' is given a `class` name of `d3js` and the text 'd3noob.org' is given a `class` name of `d3noob`.

Then when we call the `update` function each of the two text elements is adjusted seperately by selecting each based on the `class` name that was applied in the initial setup;

```
function update(nAngle) {

// adjust the range text
  d3.select("#nAngle-value").text(nAngle);
  d3.select("#nAngle").property("value", nAngle);

  // adjust d3.js text
  holder.select("text.d3js")
    .attr("transform", "translate(300,55) rotate("+nAngle+")");

  // adjust d3noob.org text
  holder.select("text.d3noob")
    .attr("transform", "translate(300,130) rotate("+(360 - nAngle)+")");
}
```

So the 'd3.js' text is selected using `text.d3js` and 'd3noob.org' is selected using `text.d3noob`. That's a pretty neat trick and a good lesson for applying specific transformations to specific objects.

Add an HTML table to your graph

So graphs and graphics are D3's bread and butter you'd think. Hmm…

Well yes and no.

Yes D3 has extraordinary powers for presenting and manipulating images in a web page. But if you've read through the entirety of the d3.js main site (haven't we all) you will recall that D3 actually stands for Data Driven Documents. It's not necessarily about the pretty pictures and the swirling cascade of colour. It's about generating something in a web browser based on data.

This transitions nicely into consideration of adding a table of information that can accompany your graph (it could just as easily (or easier) stand alone, but for the sake of continuity, we'll use the graph).

What we'll do is add the data that we've used to make our graph under the graph itself. To make sure that it's all nicely aligned, we'll place it in a table.

It should end up looking a little like this (and this has been cropped slightly at the bottom to avoid expanding the page with rows of numbers / dates).

date	close
1-May-12	58.13
30-Apr-12	53.98
27-Apr-12	67
26-Apr-12	89.7
25-Apr-12	99
24-Apr-12	130.28
23-Apr-12	166.7

Basic graph with a table of data

The code was drawn from an example provided by Shawn Allen[151] on Google Groups[152]. In fact, the post itself is an excellent one if you are considering creating a table straight from a csv file.

[151] http://jsfiddle.net/7WQjr/
[152] http://stackoverflow.com/questions/9268645/d3-creating-a-table-linked-to-a-csv-file

HTML Tables

I'm walking a fine line here since I have a remarkably small amount of knowledge on HTML tables. So I'll try to provide a brief overview as I understand it and as I see it represented in the code below, but for a far fuller explanation, take a look at some great work by Peter Cook here[153] or let Google be your friend.

Tables are made up of rows, columns and data (that goes in each cell). All you need to do to successfully place a table on a web page is to lay out the rows and columns in a logical sequence using the appropriate HTML tags and you're away.

For example here's the total HTML code for a web page to display a simple table;

```
<!DOCTYPE html>
<body>
    <table border="1">
        <tr>
            <th>Header 1</th>
            <th>Header 2</th>
        </tr>
        <tr>
            <td>row 1, cell 1</td>
            <td>row 1, cell 2</td>
        </tr>
        <tr>
            <td>row 2, cell 1</td>
            <td>row 2, cell 2</td>
        </tr>
    </table>
</body>
```

This will result in a table that looks a little like this in a web browser;

Header 1	Header 2
row 1, cell 1	row 1, cell 2
row 2, cell 1	row 2, cell 2

The entire table itself is enclosed in `<table>` tags. Each row is enclosed in `<tr>` tags. Each row has two items which equate to the two columns. Each piece of data for each cell is enclosed in a `<td>` tag except for the first row, which is a header and therefore has a special tag `<th>` that denotes it as a header making it bold and centred. For the sake of ease of viewing we have told the table to place a border around each cell and we do this in the first `<table>` tag with the `border="1"` statement (although in this book view it may be absent).

[153] http://prcweb.co.uk/lab/selection/

The good news is that you don't need to fully understand all this, but it will help with the explanation of what we're doing in the code below.

There are three main things you need to do to the basic line graph to get your table to display.

1. Add some CSS
2. Add some table building d3.js code
3. Make a small but cunning change…

There is a copy of the code and the data file for this example at github[154] and in the code samples bundled with this book (simple-graph-plus-table.html and data.csv). A live example can be found on bl.ocks.org[155].

First the CSS

This just helps the table with formatting and making sure the individual cells are spaced appropriately;

```
td, th {
    padding: 1px 4px;
}
```

This sets a padding of 1 px around each cell and 4 px between each column.

Feel free to play with the figures to suit your application, I've just set them there because I thought they looked appropriate.

I've placed this portion of CSS at the end of our `<style>` section.

Now the d3.js code

Oki doki… Hopefully you have a loose understanding of the html layout of a table as explained above, but if not you can always go with the 'it just works' approach.

Here's what we should add into our simple graph example;

[154] https://gist.github.com/d3noob/473f0cf66196a008cf99
[155] http://bl.ocks.org/d3noob/473f0cf66196a008cf99

```
function tabulate(data, columns) {
    var table = d3.select("body").append("table")
            .attr("style", "margin-left: 250px"),
        thead = table.append("thead"),
        tbody = table.append("tbody");

    // append the header row
    thead.append("tr")
        .selectAll("th")
        .data(columns)
        .enter()
        .append("th")
            .text(function(column) { return column; });

    // create a row for each object in the data
    var rows = tbody.selectAll("tr")
        .data(data)
        .enter()
        .append("tr");

    // create a cell in each row for each column
    var cells = rows.selectAll("td")
        .data(function(row) {
            return columns.map(function(column) {
                return {column: column, value: row[column]};
            });
        })
        .enter()
        .append("td")
        .attr("style", "font-family: Courier")
            .html(function(d) { return d.value; });

    return table;
}

// render the table
 var peopleTable = tabulate(data, ["date", "close"]);
```

And we should take care to add it into the code at the end of the portion where we've finished drawing the graph, but before the enclosing curly and regular brackets that complete the portion of the graph that has loaded our data.csv file. This is because we want our new piece of code to have access to that data and if we place it after those brackets it won't know what data to display.

So, right about here;

```
        // Add the Y Axis
        svg.append("g")
            .attr("class", "y axis")
            .call(yAxis);
                                    // <= Add the code right here!
});
```

Now, we're going to break with tradition a bit here and examine what our current state of code produces. Then we're going to explain something different. *THEN* we're going to come back and explain the code…

Check it out…

date	close
Tue May 01 2012 00:00:00 GMT+1200 (New Zealand Standard Time)	58.13
Mon Apr 30 2012 00:00:00 GMT+1200 (New Zealand Standard Time)	53.98
Fri Apr 27 2012 00:00:00 GMT+1200 (New Zealand Standard Time)	67
Thu Apr 26 2012 00:00:00 GMT+1200 (New Zealand Standard Time)	89.7
Wed Apr 25 2012 00:00:00 GMT+1200 (New Zealand Standard Time)	99
Tue Apr 24 2012 00:00:00 GMT+1200 (New Zealand Standard Time)	130.28

Woah! What happened to the date?

Not quite as we has originally envisaged?

Indeed, the date has taken it upon itself to expand from a relatively modest format of day-abbreviated month-two digit year (30-Apr-12) to a behemoth of a thing (Mon Apr 30 2012 00:00:00 GMT+1200 (New Zealand Standard Time)) that we certainly didn't intend, let alone have in our data.csv file.

What's going on here?

Well, To be perfectly frank, I'm not entirely sure. But this is what I'm going to propose. The JavaScript code recognises and deals with the 'date' variable as being a date/time. So that when we proceed to display the variable on the screen, the browser says, "this is a date / time formatted piece of data, therefore it must be formatted in the following way". I had a play with a few ideas to correct it via an HTML formatting instruction, but drew a blank and then I stumbled on another way to solve the problem. Hence the third small but cunning change to our original code.

A small but cunning change...

So... Our table has decided to develop a mind of it's own and format the date time as it sees fit. Well fair enough (I for one welcome our web time formatting overlords). So how do we convince it to display the values in their natural form?

Well, one solution that we could employ is to not tell the JavaScript that our `date` value in the data is actually time. In that condition, the code should treat the values as an ordinary string and print it directly as it appears.

The good news is that this is pretty easy to do. Where originally we had a block of data that consisted of `date` and `close`, all at different times, we will now add a new variable called `date1` which will be the variable that we convert to a time and draw the graph with. Leaving `date` to be the text string that will be printed in our table.

How to do it?

It's actually remarkably easy. Just change the following lines in the basic line graph code to amend `date` to `date1` and you're good to go.

```
.x(function(d) { return x(d.date1); })
```

```
d.date1 = parseDate(d.date);
```

```
x.domain(d3.extent(data, function(d) { return d.date1; }));
```

The middle line is probably the most significant, since this is the point where we declare `date1`, assign a time format and bring a new column of data into being. The others simply refer to the data.

So we'll make those small changes and now we can return to explain the d3.js code...

Explaining the d3.js code (reloaded).

So back we come to explain what is going on in the d3.js code that we presented a page or two back. Obviously it's a fairly large chunk, and we can first break it down into two chunks. The first chunk we'll look at is in fact the last part of the code that look like this;

```
// render the table
 var peopleTable = tabulate(data, ["date", "close"]);
```

This portion simply calls the `tabulate` function using the `date` and `close` columns of our `data` array. Simply add or remove whichever columns you want to appear in your table (so long as they are in your data.csv file) and they will be in your table. The tabulate function makes up all of the other part of the added code. So we come to the first block of the tabulate function;

```
function tabulate(data, columns) {
    var table = d3.select("body").append("table")
            .attr("style", "margin-left: 250px"),
        thead = table.append("thead"),
        tbody = table.append("tbody");
```

Here the tabulate function is declared (`function tabulate`) and the variables that the function will be using are specified (`(data, columns)`). In our case `data` is of course our `data` array and `columns` refers to `["date", "close"]`.

The next line appends the table to the `body` of the web page (so it will occur just under the graph in this case). The I do something just slightly sneaky. The line `.attr("style", "margin-left: 250px"),` is actually not the code that was used to produce the table with the huge date/ time formatted info on. I deliberately used `.attr("style", "margin-left: 0px"),` for the huge date / time table since it's job is to indent the table by a specified amount from the left hand side of the page. And since the huge date time values would have pushed the table severely to the right, I cheated and used `0` instead of `250`. For the purposes of the final example where the date / time values are formatted as expected, `250` is a good value.

The next two lines declare the functions we will use to add in the header cells (since they use the `<th>` tags for content) and the cells for the main body of the table (they use `<td>`).

The next block of code adds in the header row;

```
    thead.append("tr")
        .selectAll("th")
        .data(columns)
        .enter()
        .append("th")
            .text(function(column) { return column; });
```

Here we first append a row tag (`<tr>`), then we gather all the `columns` that we have in our function (remember they were `["date", "close"]` and add them to our row using header tags (`<th>`).

The next block of code assigns the `row` variable to return (append) a row tag (`<tr>`) whenever it's called ...

```
    var rows = tbody.selectAll("tr")
        .data(data)
        .enter()
        .append("tr");
```

... and it is in the following block of code...

```
var cells = rows.selectAll("td")
    .data(function(row) {
        return columns.map(function(column) {
            return {column: column, value: row[column]};
        });
    })
    .enter()
    .append("td")
    .attr("style", "font-family: Courier")
        .html(function(d) { return d.value; });
```

... where we select each row that we've added (`var cells = rows.selectAll("td")`). Then the following five lines works out from the intersection of the row and column which piece of data we're looking at for each cell.

Then the last four lines take that piece of data (`d.value`) and wrap it in table data tags (`<td>`) and place it in the correct cell as HTML.

It's a very neat piece of code and I struggle to get my head around it, but that doesn't mean that I can't appreciate the cleverness of it :-).

Wrap up

So there we have it. Hopefully enough to explain what is going on and perhaps also enough to convince ourselves that D3 is indeed more than just pretty pictures. It's all about the Data Driven Documents.

There is a copy of the code and the data file for this example at github[156] and in the code samples bundled with this book (simple-graph-plus-table.html and data.csv). A live example can be found on bl.ocks.org[157]

[156] https://gist.github.com/d3noob/473f0cf66196a008cf99
[157] http://bl.ocks.org/d3noob/473f0cf66196a008cf99

More table madness: sorting, prettifying and adding columns

When we last left our tables they were happily producing a faithful list of the data points that we had in our graph.

But what if we wanted more?

From the original contributors that bought you tables (Shawn Allen[158] on Google Groups[159]) and some neat additions from Christophe Viau[160] comes extra coolness that I didn't include in the previous example :-).

There is a copy of the code and the data file for this example at github[161] and in the code samples bundled with this book (simple-graph-plus-table-plus-addins.html and data2.csv). A live example can be found on bl.ocks.org[162].

Add another column of information:

Firstly, lets add another column of data to our table. To do this we want to have something extra in our csv file to use, so let's resurrect our old friend data2.csv that we used for the graph with two lines previously. All we have to do to make this a reality is change the reference that loads `data.csv` to `data2.csv` here;

```
d3.csv("data2.csv", function(error, data) {
```

This makes the assumption that you still have the data2.csv file in place. If not, rush away and get it from the download of code samples from Leanpub.

From here (and as promised in the previous chapter), it's just a matter of adding in the extra column you want (in this case it's the `open` column) like so;

```
var peopleTable = tabulate(data, ["date", "close", "open"]);
```

[158] http://jsfiddle.net/7WQjr/
[159] http://stackoverflow.com/questions/9268645/d3-creating-a-table-linked-to-a-csv-file
[160] http://christopheviau.com/d3_tutorial/
[161] https://gist.github.com/d3noob/5d47df5374d210b6f651
[162] http://bl.ocks.org/d3noob/5d47df5374d210b6f651

date	close	open
1-May-12	58.13	34.12
30-Apr-12	53.98	45.56
27-Apr-12	67	67.89
26-Apr-12	89.7	78.54
25-Apr-12	99	89.23
24-Apr-12	130.28	99.23

Table with extra column

Yes, if you're wondering, I have cheated slightly and changed the table indent to make it look slightly prettier.

So can we go further?

You know we can...

In the section where we get our data and format it, lets add another column to our array in the form of a difference between the `close` value and the `open` value (and we'll call it `diff`).

```
d3.csv("data2.csv", function(error, data) {
        data.forEach(function(d) {
                d.date1 = parseDate(d.date);
                d.close = +d.close;
                d.open = +d.open;  //  <= added this for tidy house keeping
                d.diff = Math.round(( d.close - d.open ) * 100 ) / 100;
        });
```

(the `Math.round` function is to make sure we get a reasonable figure to display, otherwise it tends to get carried away with decimal places)

So now we add in our new column (`diff`) to be tabulated;

```
var peopleTable = tabulate(data, ["date", "close", "open", "diff"]);
```

Table with extra extra column

And yes, I changed the table indent again. I am a serial offender and will continue to change it to suit.

Sorting on a column

So now with our four columns of awesome data, it turns out that we're really interested in the ones that have the highest `close` values. So we can sort on the `close` column by adding the following lines directly after the line where we declare the `peopleTable` function (which I will include in the code snipped below for reference).

```
var peopleTable = tabulate(data, ["date", "close", "open", "diff"]);

peopleTable.selectAll("tbody tr")
        .sort(function(a, b) {
                return d3.descending(a.close, b.close);
        });
```

Which works magnificently;

Table sorted descending on 'close'

Prettifying (actually just capitalising the header for each column)

Just a little snippet that capitalises the headers for each row to make them look slightly more authoritative.

Add the following lines of code directly below the block that you just added for sorting the table;

```
peopleTable.selectAll("thead th")
        .text(function(column) {
                return column.charAt(0).toUpperCase() + column.substr(1);
        });
```

This is quite a tidy little piece of script. You can see it selecting the headers (`selectAll("thead th")`), then the first character in each header (`column.charAt(0)`), changing it to upper-case (`.toUpperCase()`) and adding it back to the rest of the string (`+ column.substr(1)`).

With the ultimate result...

Date	Close	Open	Diff
9-Apr-12	636.23	350.45	285.78
5-Apr-12	633.68	410.23	223.45
3-Apr-12	629.32	460.34	168.98
10-Apr-12	628.44	310.45	317.99
11-Apr-12	626.2	212.67	413.53

Table with capitalised first characters in headers

Add borders

Sure our table looks nice and neatly arranged, but would a border look better?

Well, here's one way to do it;

All we need to do is add a border style to our table by adding in this line here;

```
function tabulate(data, columns) {
    var table = d3.select("body").append("table")
            .attr("style", "margin-left: 200px") // <= Remove the comma
            .style("border", "2px black solid"), // <= Add this line in
        thead = table.append("thead"),
        tbody = table.append("tbody");
```

(don't forget to move the comma from the end of the `margin-left` line)

And the result is a tidy black border.

Table with a border

OK, so what about the individual cells?

No problem.

If we remember back to our CSS that we added in, we'll just tell each cell that we want a 1 pixel border buy amending the CSS for our table to this;

```
td, th {
    padding: 1px 4px;
    border: 1px black solid;
}
```

So now each cell has a slightly more subtle border like this;

Date	Close	Open	Diff
9-Apr-12	636.23	350.45	285.78
5-Apr-12	633.68	410.23	223.45
3-Apr-12	629.32	460.34	168.98
10-Apr-12	628.44	310.45	317.99
11-Apr-12	626.2	212.67	413.53

Table with cells with individual borders

Yikes! Not quite as subtle as I would have expected. I suppose it's another example of the code actually doing what you asked it to do. No problem, `border-collapse` to the rescue. Add the following line into here;

```
function tabulate(data, columns) {
    var table = d3.select("body").append("table")
            .attr("style", "margin-left: 200px")
            .style("border-collapse", "collapse")   // <= Add this line in.
            .style("border", "2px black solid"),
        thead = table.append("thead"),
        tbody = table.append("tbody");
```

How does that look?

Table with cells with collapsed borders

Ahh.... Much more refined.

The `border-collapse` style tells the table to overlap each cell's borders, rather than treat them as discrete entities. So in this case it looks a bit better.

Adding web links to d3.js objects

The idea with this tip / trick is to be able to add a 'link' to an object so that when you click on it, it takes you to a web page that we will pre-define.

We are going to generate a simple rectangle with some text and look at linking from the rectangle and the text separately and with some fanciness at the end :-).

The end result will be something that looks a little like this;

Objects with links

(Notice the little pointing finger at the bottom that would indicate that there actually *is* a link there.)

The code that we will use as a starting point is this simple example that draws a green rectangle and overlays some text on it;

```
<!DOCTYPE html>
<meta charset="utf-8">

<body>

<!-- load the d3.js library -->
<script src="http://d3js.org/d3.v3.min.js"></script>

<script>

var width = 449;
var height = 249;
var word = "gongoozler";

var holder = d3.select("body")
      .append("svg")
      .attr("width", width)
      .attr("height", height);

// draw a rectangle
holder.append("rect")
    .attr("x", 100)
    .attr("y", 50)
    .attr("height", 100)
    .attr("width", 200)
    .style("fill", "lightgreen")
    .attr("rx", 10)
    .attr("ry", 10);
```

```
// draw text on the screen
holder.append("text")
    .attr("x", 200)
    .attr("y", 100)
    .style("fill", "black")
    .style("font-size", "20px")
    .attr("dy", ".35em")
    .attr("text-anchor", "middle")
    .text(word);

</script>

</body>
```

There's nothing too spectacular about the file. There's a little bit of styling and tweaking of attributes, but nothing too extreme. The only slightly 'odd' part would be defining the word that is printed out as a variable (`var word = "gongoozler";`) and then adding it as a variable (`.text(word);`) instead of just putting the word directly in there (which we could do like this `.text("gongoozler");`). We're going to do this deliberately to explore additional options for making our links a little more dynamic.

It's all about the 'a' and the 'xlink'

The `<a>` tag in an HTML file defines a hyperlink[163]. Items bounded by an `<a>` tag will become a link to another web address. So what we will do is create an `<a>` tag and then append our d3.js, svg object to it.

Of course as well as including a link, we need to tell it where to go. We do this by setting the `xlink:href` attribute for our tag to point to a specific page. Xlink is short for XML Linking Language and it is used to create hyperlinks in XML documents. In our case we will be defining the link that we will want our user to go to.

Adding in the links

The following is the adjusted code for our rectangle that adds in the `<a>` tag with the `xlink:href` attribute.

[163] http://www.w3schools.com/html/html_links.asp

```
holder.append("a")
    .attr("xlink:href", "http://en.wikipedia.org")
    .append("rect")
    .attr("x", 100)
    .attr("y", 50)
    .attr("height", 100)
    .attr("width", 200)
    .style("fill", "lightgreen")
    .attr("rx", 10)
    .attr("ry", 10);
```

It's important to append the link before the object (otherwise it won't work) but other than that, it's a pretty simple job.

The only fly in the ointment is that while we now have a rectangle that links to Wikipedia, if we hover our mouse over the text, we lose our link (since we haven't told the text to link anywhere).

We can remedy that by doing exactly the same thing with the text element;

```
holder.append("a")
    .attr("xlink:href", "http://en.wikipedia.org/wiki/"+word)
    .append("text")
    .attr("x", 200)
    .attr("y", 100)
    .style("fill", "black")
    .style("font-size", "20px")
    .attr("dy", ".35em")
    .attr("text-anchor", "middle")
    .text(word);
```

The only slight difference here is that we have used the address for Wikipedia as our base and added the variable for our word to the end of it so that the resulting web address takes us to Wikipedia and the specific page for the word 'gongoozler'. Hopefully this will indicate that if we had a set of variables in an array we would make our links a little more dynamic.

Making the mouse pointer ignore an object

So in theory we're done, but in practice this has been a slightly crude method for adding what should be a *single* link to two objects when we should be able to accomplish it by defining the link once.

What we could do as an alternative to linking both the rectangle and the text using two separate links is to make the mouse ignore the text and have it rely solely on the rectangle. We can do this using the `pointer-events` style when drawing our text. By setting it to `none` we are instructing our mouse to ignore any potential interaction with the text when it hovers over it and instead the pointer will register the link on the rectangle below it.

The code for the text therefore becomes…

```
holder.append("text")
    .attr("x", 200)
    .attr("y", 100)
    .style("fill", "black")
    .style("font-size", "20px")
    .attr("dy", ".35em")
    .attr("text-anchor", "middle")
    .style("pointer-events", "none")
    .text(word);
```

And as you can see from the image below, the pointer will happily ignore the text while reading the link from the rectangle.

Objects with links

The complete code for this example is available in the appendices and a live version can be found on bl.ocks.org[164] and GitHub[165]. The code is also in the downloadable files available from Leanpub with the book in a file called 'xlink-rect&pointerevents.html'.

[164] http://bl.ocks.org/d3noob/8150631
[165] https://gist.github.com/d3noob/8150631

Understanding JavaScript Object Notation (JSON)

One of the most useful things you might want to learn when understanding how to present your data with D3 is how to *structure* your data so that it is easy to use.

As explained earlier in the book, there are several different types of data that can be requested by D3 including text, Extensible Markup Language (xml), HyperText Markup Language (html), Comma Separated Values (csv), Tab Separated Values (tsv) and JavaScript Object Notation (json).

Comma separated values and tab separated values are fairly well understood forms of data. They are expressed as rows and columns of information that are separated using a known character. While these forms of data are simple to understand, it is not easy to incorporate a hierarchy structure to the data, and when you try, it isn't natural and makes managing the data difficult.

JavaScript Object Notation (JSON) presents a different mechanism for storing data. A light weight description could read "JSON is a text-based open standard designed to present human-readable data. It is derived from the JavaScript scripting language, but it is language and platform independent."

Unfortunately, when I first started using JSON, I struggled with the concept of how it was structured, in spite of some fine descriptions on the web (start with http://www.json.org/[166] in my humble opinion). So the following is how I came to think of and understand JSON.

Fair Warning: This advice is no substitute for the correct explanation of the topic of data structures that I'm sure you could receive from a reputable educational site or institution. It's just the way I like to think of it :-). It's also just the way that I *started* to understand JSON. There is plenty to learn and understand once you grasp the basics. So this isn't a complete guide. Just the beginnings.

In the following steps we'll go through a process that (hopefully) demonstrates that we can transform identifiers that would represent the closing price for a stock of 58.3 on 2013-03-14 into more traditional x,y coordinates.

I think of data as having an identifier and a value.

```
identifier: value
```

If a point on a graph is located at the x,y coordinates 150,25 then the identifier 'x' has a value 150.

```
"x": 150
```

If the x axis was a time-line, the true value for 'x' could be "2013-03-14".

```
"x": "2013-03-14"
```

[166] http://www.json.org/

This example might look similar to those seen by users of d3.js, since if we're using date / time format we can let D3 sort out the messy parts like what coordinates to provide for the screen.

And there's no reason why we couldn't give the 'x' identifier a more human readable label such as "date". So our data would look like;

```
"date": "2013-03-14"
```

This is only one part of our original x,y = 150,25 data set. The same way that the x value represented a position on the x axis that was really a date, the y value represents a position on the y axis that is really another number. It only gets converted to 25 when we need to plot a position on a graph at 150,25. If the 'y' component represents the closing price of a stock we could take the same principles used to transform…

```
"x": 150
```

… into …

```
"date": "2013-03-14"
```

… to change ….

```
"y": 25
```

… into …

```
"close": 58.3
```

This might sound slightly confusing, so try to think of it this way. We want to plot a point on a graph at 150,25, but the data that this position is derived from is really "2013-03-14", 58.3. D3 can look after all the scaling and determination of the range so that the point gets plotted at 150,25 and our originating data can now be represented as;

```
"date": "2013-03-14", "close": 58.3
```

This represents two separate pieces of data. Each of which has an identifier ("date" or "close") and a value ("2013-03-14" and 58.3)

If we wanted to have a series of these data points that represented several days of closing prices, we would store them as an array of identifiers and values similar to this;

```
{ "date": "2013-03-14", close: 58.13 },
{ "date": "2013-03-15", close: 53.98 },
{ "date": "2013-03-16", close: 67.00 },
{ "date": "2013-03-17", close: 89.70 },
{ "date": "2013-03-18", close: 99.00 }
```

Each of the individual elements of the array is enclosed in curly brackets and separated by commas.

I am making the assumption that you are familiar with the concept of what an 'array' is. If this is an unfamiliar word, in the context of data, then I strongly recommend that you do some Goggling to build up some familiarity with the principle.

Now that we have an array, we can apply the same rules to it as we did the the item that had a single value. We can give it an identifier all of its own. In this case we will call it "data". Now we can use our identifier: value analogy to use "data" as the identifier and the array as the value.

```
{ "data": [
  { "date": "2013-03-14", close: 58.13 },
  { "date": "2013-03-15", close: 53.98 },
  { "date": "2013-03-16", close: 67.00 },
  { "date": "2013-03-17", close: 89.70 },
  { "date": "2013-03-18", close: 99.00 }
] }
```

The array has been enclosed in square brackets to designate it as an array and the entire identifier: value sequence has been encapsulated with curly braces (much the same way that the subset "date", "close" values were enclosed with curly braces.

If we try to convey the same principle in a more graphical format, we could show our initial identifier and value for the x component like so;

Single identifier and value

The we can add our additional component for the y value;

Single identifier and value

We can then add several of these combinations together in an array;

Single identifier and value

Then the array becomes a value for another identifier "data";

Single identifier and value

More complex JSON files will have multiple levels of identifiers and values arranged in complex hierarchies which can be difficult to interpret. However, laying out the data in a logical way in a text file is an excellent way to start to make sense of the data.

Using the Yahoo Query Language (YQL) to get data.

One of the things that I find frustrating is difficulty getting data to play with. Often I will find myself thinking "It would be neat to see if this technique works" but then I spend a couple of hours looking for data that will be appropriate.

Another difficulty is that access to dynamic data is also problematic. D3.js really shines when displaying information that is changing and interactive. Finding data that is changing and which you can interact with is not easy.

Then, when you do come across a web site that has an interesting data set, accessing that data becomes programmatically difficult because of restrictions in accessing information that crosses domains. A solution to this problem (and there are a few) is to have a data set that is in a domain that permits Cross-Origin Resource Sharing[167] (CORS). These are not as common as you might hope as it presents security concerns that need to be addressed.

One resource that I have come across is particularly useful for solving the problems outlined above is the Yahoo! Developer Network[168] which operates a service where you can write a query in an SQL type fashion that will return data in json (or others) format. This query can be formed into an http request so that you can simply paste a generated URL directly into your browsers URL bar and it will return the requested data. Better than that, because it supports CORS (and because d3.js can manage a CORS request without breaking a sweat) all you need to do is put the URL into you code (as a `d3.json` call for example) and you will be up and running.

In the 'Examples' chapter there will be several examples that use YQL queries to retrieve data as is is a great resource.

YQL by example

I will use an example of looking for current weather information for Wellington New Zealand from the Weather Underground[169] (wunderground).

In this example, we're interested in retrieving a JSON 'blob' that contains data on the weather conditions (temperature, wind speed / direction, humidity etc) for a specific location.

Start at the page for the console (http://developer.yahoo.com/yql/console/[170]). On the left there is a range of data 'topics that you can choose from. We will want to access one of the community tables, so select 'Show Community Tables' and scroll down the list until you find 'wunderground' and 'wunderground.currentobservation'

[167] http://en.wikipedia.org/wiki/Cross-origin_resource_sharing
[168] http://developer.yahoo.com/yql/
[169] http://www.wunderground.com/
[170] http://developer.yahoo.com/yql/console/

Select Community Tables and wunderground.currentobservation

This will bring up a query in the console that is looking for the current weather from San Francisco International airport.

Query for weather from San Francisco

For our example we want to be a bit more specific, so we replace the 'SFO' 'location' with 'Wellington, New Zealand'.

If you then click on the 'Test' button the query will run and results will be presented in the box below it;

Formatted View | Tree View | View ▾ | wunderground.currentobservation


```
},
"station_id": "NZWN",
"observation_time": "Last Updated on March 16, 7:30 AM NZDT",
"observation_time_rfc822": "Sat, 15 Mar 2014 18:30:00 GMT",
"observation_epoch": "1394908200",
"local_time": "March 16, 7:45 AM NZDT",
"local_time_rfc822": "Sat, 15 Mar 2014 18:45:52 GMT",
"local_epoch": "1394909152",
"weather": "Clear",
"temperature_string": "70 F (21 C)",
"temp_f": "70",
"temp_c": "21",
"relative_humidity": "78%",
"wind_string": "From the NE at 14 MPH ",
"wind_dir": "NE",
"wind_degrees": "50",
"wind_mph": "14",
"wind_gust_mph": null,
```


JSON weather from San Francisco

If that looks like the results you were expecting, fantastic!

The next gem from YQL is the REST[171] query that is automatically generated at the bottom of the page;

THE REST QUERY How do I use this?

http://query.yahooapis.com/v1/public/yql?q=select%20*%20from%20wunderground.curren

REST Query

If you select this query and past it in your URL bar, you will have the JSON data returned into your browser.

At this point it might seem a bit confusing, and if you're unfamiliar with how JSON based data sets are structured it will DEFINITELY be confusing (You will need to do some research to get up to speed there). My advice is to first understand a bit about JSON and then examine the output from the REST query in the developer console on your browser (this is a fairly long topic in itself, so I will not cover it here sorry). Additionally check out the 'Examples' chapter for real world usage.

If you are comfortable with understanding how the data can be encoded, it is then just a matter of including the REST query in your call when selecting data using `d3.json` and you will be away.

Please check out the examples I will include in the 'Examples' chapter for a deeper look at the use of the data with d3.js (This will also help with understanding the JSON structure).

[171] http://en.wikipedia.org/wiki/Representational_state_transfer

Using Plunker for development and hosting your D3 creations.

Recently Mike Bostock recommended 'Plunker' (http://plnkr.co/[172]) as a tool for saving work online for collaboration and sharing. Although I had a quick look, I didn't quite 'get it' and although it looked like something that I should be interested in, I (foolishly) moved on to other things.

Quite frankly I should have persevered.

Plunker is awesome.

So what can it do for you?

Well, in short, this gives you a place to put your graphs on the web without the hassle of needing a web server as well as allowing others to view and collaborate! There are some limitations to hosting graphs in this environment, but there's no denying that for ease of use and visibility to the outside world, it's outstanding!

Time for an example. I'll try to go through the process of implementing the simple graph example on Plunker.

So it's as simple as going to http://plnkr.co/edit/[173]

Plunker editing page

What you're seeing here is an area where you can place your entire HTML code. So let's replace the 11 lines of the place holder code with the simple graph example (just copy and paste it in there over the top of the current 11 lines);

Now, there are two important things we have to do before it will work.

1. We need to tell he script where to find d3.js
2. We need to make our data accessible

Helping the script find d3.js is nice and easy. Just replace this line in your plunk;

[172] http://plnkr.co/
[173] http://plnkr.co/edit/

```
<script type="text/javascript" src="d3/d3.v3.js"></script>
```

...with this line...

```
<script src="http://d3js.org/d3.v3.min.js"></script>
```

That will allow your plunk to use the version of d3.js that is hosted on d3js.org (it uses the minimised version (which is why it has the 'min' in it), but never fear, it's still d3, just distilled to enhance the flavour :-)).

Making our data available is only slightly more difficult.

In experimenting with Plunker, I found that there appears to be something 'odd' about accessing tab separated values (in the data.tsv file), however, D3 to the rescue! We can simply use Comma Separated Values (csv) instead.

We will host our data.csv file on plunker as well and there is built in functionality to let us do it.

Create a new file

In the top left hand corner, beside the 'FILES' note, there is a '+NEW...' section. Clicking on this will allow you to create another file that will exist with your plunk for its use, so let's do that.

This will open a dialogue box that will ask you to name your new file.

Name your file

Enter the name data.csv.

Now another file has appeared under the 'Files' heading called data.csv. Click on it.

The empty data.csv file

This now shows us a blank file called data.csv, so now open up your data.csv file in whatever editor you're using (I don't think a spreadsheet program is going to be a good idea since I doubt that it will maintain the information in a textual form as we're wanting it to do. So it's Geany for me). Copy the contents of your local data.csv file and paste it into the new plunker data.csv file.

So now we have our data in there we need to tell our JavaScript where it is. So go back to the 'index.html' file (which is our simple graph code) and edit the line which finds the data.tsv file from this

```
d3.tsv("data/data.tsv", function(error, data) {
```

... to this ...

```
d3.csv("data.csv", function(error, data) {
```

Because we're using relative addressing, and plunker stores the files for the graphing script and the data side by side, we just removed the portion of the address that told our original code to look in the 'data' directory and told it to look in the current directory. And that should be that!

Now if you look on the right hand side of the screen, there is a little eye icon. If you click on it, it opens up a preview window of your file in action and viola!

Preview your graph

If the graph doesn't appear, go through the steps outlined above and just check that the edits are made correctly. Unfortunately I haven't found a nice mechanism for troubleshooting inside Plunker yet (not like using F12 on Chrome).

But wait! There's more!

If you now click on the 'Save' button at the top of the screen, you will get some new button options.

One of them is the orange one for showing off your work.

Show off your work

If you click on this, it will present you with several different options.

Preview your graph

The first one is a link that will give others the option to collaborate on the script.

The second is a link that will allow others to preview the work; http://embed.plnkr.co/QSCkG8Rf2qFgrCqq7Vfn[174]

The last will allow you to embed your graph in a separate web page somewhere. Which I've tested with blogger and seems to work really well! (see image below).

Plunker iframe inserted in a blog post

[174] http://embed.plnkr.co/QSCkG8Rf2qFgrCqq7Vfn

So, I'm impressed, Nice work by Plunker and it's creator Geoff Goodman.

Bar Charts

A bar chart is a visual representation using either horizontal or vertical bars to show comparisons between discrete categories. There are a number of variations of bar charts including stacked, grouped, horizontal and vertical.

There is a wealth of examples of bar charts on the web, but I would recommend a visit to the D3.js gallery[175] maintained by Christophe Viau as a starting point to get some ideas.

We will work through a simple vertical bar chart that uses a value on the y axis and date values on the x axis.

The end result will look like this;

Bar chart

The data

The data for this example will be sourced from an external csv file named `bar-data.csv`. It consists of a column of dates in year-month format and it's contents are as follows;

```
date,value
2013-01,53
2013-02,165
2013-03,269
2013-04,344
2013-05,376
2013-06,410
2013-07,421
2013-08,405
2013-09,376
```

[175] http://christopheviau.com/d3list/gallery.html#visualizationType=bar

```
2013-10,359
2013-11,392
2013-12,433
2014-01,455
2014-02,478
```

The code

The full code listing for the example we are going to work through is as follows;

```
<!DOCTYPE html>
<meta charset="utf-8">

<head>
        <style>

        .axis {
          font: 10px sans-serif;
        }

        .axis path,
        .axis line {
          fill: none;
          stroke: #000;
          shape-rendering: crispEdges;
        }

        </style>
</head>

<body>

<script src="http://d3js.org/d3.v3.min.js"></script>

<script>

var margin = {top: 20, right: 20, bottom: 70, left: 40},
    width = 600 - margin.left - margin.right,
    height = 300 - margin.top - margin.bottom;

// Parse the date / time
var        parseDate = d3.time.format("%Y-%m").parse;

var x = d3.scale.ordinal().rangeRoundBands([0, width], .05);
```

```
var y = d3.scale.linear().range([height, 0]);

var xAxis = d3.svg.axis()
    .scale(x)
    .orient("bottom")
    .tickFormat(d3.time.format("%Y-%m"));

var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left")
    .ticks(10);

var svg = d3.select("body").append("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
  .append("g")
    .attr("transform",
          "translate(" + margin.left + "," + margin.top + ")");

d3.csv("bar-data.csv", function(error, data) {

    data.forEach(function(d) {
        d.date = parseDate(d.date);
        d.value = +d.value;
    });

  x.domain(data.map(function(d) { return d.date; }));
  y.domain([0, d3.max(data, function(d) { return d.value; })]);

  svg.append("g")
      .attr("class", "x axis")
      .attr("transform", "translate(0," + height + ")")
      .call(xAxis)
    .selectAll("text")
      .style("text-anchor", "end")
      .attr("dx", "-.8em")
      .attr("dy", "-.55em")
      .attr("transform", "rotate(-90)" );

  svg.append("g")
      .attr("class", "y axis")
      .call(yAxis)
    .append("text")
      .attr("transform", "rotate(-90)")
      .attr("y", 6)
```

```
      .attr("dy", ".71em")
      .style("text-anchor", "end")
      .text("Value ($)");

  svg.selectAll("bar")
      .data(data)
    .enter().append("rect")
      .style("fill", "steelblue")
      .attr("x", function(d) { return x(d.date); })
      .attr("width", x.rangeBand())
      .attr("y", function(d) { return y(d.value); })
      .attr("height", function(d) { return height - y(d.value); });

});

</script>

</body>
```

The full code for this example can be found on github[176], in the appendices of this book or in the code samples bundled with this book[177] (bar.html and bar-data.csv). A working example can be found on bl.ocks.org[178].

The bar chart explained

In the course of describing the operation of the file I will gloss over the aspects of the structure of an HTML file which have already been described at the start of the book. Likewise, aspects of the JavaScript functions that have already been covered will only be briefly explained.

The start of the file deals with setting up the document's head and body, loading the d3.js script and setting up the css in the `<style>` section.

The css section sets styling for the axes. It sizes the font to be used and make sure the lines are formatted appropriately.

[176] https://gist.github.com/d3noob/8952219
[177] https://leanpub.com/D3-Tips-and-Tricks
[178] http://bl.ocks.org/d3noob/8952219

```
.axis {
  font: 10px sans-serif;
}

.axis path,
.axis line {
  fill: none;
  stroke: #000;
  shape-rendering: crispEdges;
}
```

Then our JavaScript section starts and the first thing that happens is that we set the size of the area that we're going to use for the chart and the margins;

```
var margin = {top: 20, right: 20, bottom: 70, left: 40},
    width = 600 - margin.left - margin.right,
    height = 300 - margin.top - margin.bottom;
```

The next section of our code includes some of the functions that will be called from the main body of the code.

We have a familiar `parseDate` function with a slight twist. Since our source data for the date is made up of only the year and month, these are the only two portions of the date that need to be recognised;

```
var        parseDate = d3.time.format("%Y-%m").parse;
```

The next section declares the function to determine positioning in the x domain.

```
var x = d3.scale.ordinal().rangeRoundBands([0, width], .05);
```

The ordinal scale is used to describe a range of discrete values. In our case they are a set of monthly values. The `rangeRoundBands` operator provides the magic that arranges our bars in a graceful way across the x axis. In our example we use it to set the range that our bars will cover (in this case from `0` to the `width` of the graph) and the amount of padding between the bars (in this case we have selected `.05` which equates to approximately (depending on the number of pixels available) `5%` of the bar width.

The function to set the scaling in the y domain is the same as most of our other graph examples;

```
var y = d3.scale.linear().range([height, 0]);
```

The declarations for our two axes are relatively simple, with the only exception being to force the format of the labels for the x axis into a 'year-month' format.

```
var xAxis = d3.svg.axis()
    .scale(x)
    .orient("bottom")
    .tickFormat(d3.time.format("%Y-%m"));

var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left")
    .ticks(10);
```

The next block of code selects the `body` on the web page and appends an svg object to it of the size that we have set up with our `width`, `height` and `margin`'s.

```
var svg = d3.select("body").append("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
  .append("g")
    .attr("transform",
          "translate(" + margin.left + "," + margin.top + ")");
```

It also adds a `g` element that provides a reference point for adding our axes.

Then we begin the main body of our JavaScript. We load our csv file and then loop through it making sure that the dates and numerical values are recognised correctly;

```
d3.csv("bar-data.csv", function(error, data) {

    data.forEach(function(d) {
        d.date = parseDate(d.date);
        d.value = +d.value;
    });
```

We then then work through our x and y data and ensure that it is scaled to the domains we are working in;

```
  x.domain(data.map(function(d) { return d.date; }));
  y.domain([0, d3.max(data, function(d) { return d.value; })]);
```

Following that we append our x axis;

```
svg.append("g")
    .attr("class", "x axis")
    .attr("transform", "translate(0," + height + ")")
    .call(xAxis)
  .selectAll("text")
    .style("text-anchor", "end")
    .attr("dx", "-.8em")
    .attr("dy", "-.55em")
    .attr("transform", "rotate(-90)" );
```

This is placed in the correct position `.attr("transform", "translate(0," + height + ")")` and the text is positioned (using `dx` and `dy`) and rotated (`.attr("transform", "rotate(-90)" );`) so that it is aligned vertically.

Then we append our y axis in a similar way and append a label (`.text("Value ($)");`);

```
svg.append("g")
    .attr("class", "y axis")
    .call(yAxis)
  .append("text")
    .attr("transform", "rotate(-90)")
    .attr("y", 6)
    .attr("dy", ".71em")
    .style("text-anchor", "end")
    .text("Value ($)");
```

Lastly we add the bars to our chart;

```
svg.selectAll("bar")
    .data(data)
  .enter().append("rect")
    .style("fill", "steelblue")
    .attr("x", function(d) { return x(d.date); })
    .attr("width", x.rangeBand())
    .attr("y", function(d) { return y(d.value); })
    .attr("height", function(d) { return height - y(d.value); });
```

This block of code creates the bars (`selectAll("bar")`) and associates each of them with a data set (`.data(data)`).

We then append a rectangle (`.append("rect")`) with values for x/y position and height/width as configured in our earlier code.

The end result is our pretty looking bar chart;

Value ($)
450
400
350
300
250
200
150
100
50
0
2013-01
2013-02
2013-03
2013-04
2013-05
2013-06
2013-07
2013-08
2013-09
2013-10
2013-11
2013-12
2014-01
2014-02

Bar chart

Tree Diagrams

What is a Tree Diagram?

The 'Tree layout[179]' is not a distinct type of diagram per se. Instead, it's representative of D3's family of hierarchical layouts.

It's designed to produce a 'node-link' diagram that lays out the connection between nodes in a method that displays the relationship of one node to another in a parent-child fashion.

For example, the following diagram shows a root node (the starting position) labelled 'Top Node' which has two children (Bob: Child of Top Node and Sally: Child of Top Node). Subsequently, Bob:Child of Top Node has two dependant nodes (children) 'Son of Bob' and 'Daughter of Bob'.

Tree layout diagram

The clear advantage to this style of diagram is that describing it in text is difficult, but representing it graphically makes the relationships easy to determine.

The data required to produce this type of layout needs to describe the relationships, but this is not necessarily an onerous task. For example, the following is the data (in JSON form) for the diagram above and it shows the minimum information required to form the correct layout hierarchy.

```
{
  "name": "Top Node",
  "children": [
    {
      "name": "Bob: Child of Top Node",
      "parent": "Top Node",
      "children": [
        {
          "name": "Son of Bob",
          "parent": "Bob: Child of Top Node"
        },
        {
          "name": "Daughter of Bob",
          "parent": "Bob: Child of Top Node"
        }
      ]
    },
```

[179] https://github.com/mbostock/d3/wiki/Tree-Layout

```
      {
        "name": "Sally: Child of Top Node",
        "parent": "Top Node"
      }
    ]
  }
```

It shows each node as having a name that identifies it on the tree and, where appropriate, the children it has (as an array) and its parent.

The data shown above is arranged as a hierarchy and it is not always possible to source data that is organised so nicely. As we go through use examples for this type of diagram we will look at options for importing 'flat' data and converting it into a hierarchical form.

There is a wealth of examples of tree diagrams on the web, but I would recommend a visit to the D3.js gallery[180] maintained by Christophe Viau as a starting point to get some ideas.

In this chapter we're going to look at a very simple piece of code to generate a tree diagram before looking at different ways to adapt it. Including rotating it to be vertical, adding some dynamic styling to the nodes, importing from a flat file and from an external source. Finally we'll look at a more complex example that is more commonly used on the web that allows a user to expand and collapse nodes interactively.

[180] http://christopheviau.com/d3list/gallery.html#visualizationType=tree

A simple Tree Diagram explained

We are going to work through a simple example of the code that draws a tree diagram, This is more for the understanding of the process rather than because it is a good example of code for drawing a tree diagram. It is a very limited example that lacks any real interactivity which is one of the strengths of d3.js graphics. However, we will outline the operation of an interactive version towards the end of the chapter once we have explored some possible configuration options that we might want to make.

The graphic that we are going to generate will look like this...

Simple tree layout diagram

And the full code for it looks like this;

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">

    <title>Collapsible Tree Example</title>

    <style>

        .node circle {
          fill: #fff;
          stroke: steelblue;
          stroke-width: 3px;
        }

        .node text { font: 12px sans-serif; }

        .link {
          fill: none;
          stroke: #ccc;
          stroke-width: 2px;
        }

    </style>

  </head>

  <body>

<!-- load the d3.js library -->
```

```
<script src="http://d3js.org/d3.v3.min.js"></script>

<script>

var treeData = [
  {
    "name": "Top Level",
    "parent": "null",
    "children": [
      {
        "name": "Level 2: A",
        "parent": "Top Level",
        "children": [
          {
            "name": "Son of A",
            "parent": "Level 2: A"
          },
          {
            "name": "Daughter of A",
            "parent": "Level 2: A"
          }
        ]
      },
      {
        "name": "Level 2: B",
        "parent": "Top Level"
      }
    ]
  }
];

// ************** Generate the tree diagram         *****************
var margin = {top: 20, right: 120, bottom: 20, left: 120},
        width = 960 - margin.right - margin.left,
        height = 500 - margin.top - margin.bottom;

var i = 0;

var tree = d3.layout.tree()
        .size([height, width]);

var diagonal = d3.svg.diagonal()
        .projection(function(d) { return [d.y, d.x]; });

var svg = d3.select("body").append("svg")
        .attr("width", width + margin.right + margin.left)
```

```
        .attr("height", height + margin.top + margin.bottom)
  .append("g")
        .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

root = treeData[0];

update(root);

function update(source) {

  // Compute the new tree layout.
  var nodes = tree.nodes(root).reverse(),
          links = tree.links(nodes);

  // Normalize for fixed-depth.
  nodes.forEach(function(d) { d.y = d.depth * 180; });

  // Declare the nodesâ€¦
  var node = svg.selectAll("g.node")
          .data(nodes, function(d) { return d.id || (d.id = ++i); });

  // Enter the nodes.
  var nodeEnter = node.enter().append("g")
          .attr("class", "node")
          .attr("transform", function(d) {
                  return "translate(" + d.y + "," + d.x + ")"; });

  nodeEnter.append("circle")
          .attr("r", 10)
          .style("fill", "#fff");

  nodeEnter.append("text")
          .attr("x", function(d) {
                  return d.children || d._children ? -13 : 13; })
          .attr("dy", ".35em")
          .attr("text-anchor", function(d) {
                  return d.children || d._children ? "end" : "start"; })
          .text(function(d) { return d.name; })
          .style("fill-opacity", 1);

  // Declare the linksâ€¦
  var link = svg.selectAll("path.link")
          .data(links, function(d) { return d.target.id; });

  // Enter the links.
  link.enter().insert("path", "g")
```

```
          .attr("class", "link")
          .attr("d", diagonal);

}

</script>

  </body>
</html>
```

The full code for this example can be found on github[181], in the appendices of this book or in the code samples bundled with this book (simple-tree-diagram.html). A working example can be found on bl.ocks.org[182].

In the course of describing the operation of the file I will gloss over the aspects of the structure of an HTML file which have already been described at the start of the book. Likewise, aspects of the JavaScript functions that have already been covered will only be briefly explained.

The start of the file deals with setting up the document's head and body loading the d3.js script and setting up the css in the `<style>` section.

The css section sets styling for the circle that represents the nodes, the text alongside them and the links between them.

```
        .node circle {
          fill: #fff;
          stroke: steelblue;
          stroke-width: 3px;
        }

        .node text { font: 12px sans-serif; }

        .link {
          fill: none;
          stroke: #ccc;
          stroke-width: 2px;
        }
```

Then our JavaScript section starts and the first thing that happens is that we declare our array of data in the following code;

[181] https://gist.github.com/d3noob/8323795
[182] http://bl.ocks.org/d3noob/8323795

```
var treeData = [
  {
    "name": "Top Level",
    "parent": "null",
    "children": [
      {
        "name": "Level 2: A",
        "parent": "Top Level",
        "children": [
          {
            "name": "Son of A",
            "parent": "Level 2: A"
          },
          {
            "name": "Daughter of A",
            "parent": "Level 2: A"
          }
        ]
      },
      {
        "name": "Level 2: B",
        "parent": "Top Level"
      }
    ]
  }
];
```

As outlined at the start of the chapter, this data is encoded hierarchically in JavaScript Object Notation (JSON). Each node must have a name and either a parent or child node(s) or both. There are many examples of hierarchical data that can be encoded in this way. From the traditional parent - offspring example to directories on a hard drive or a breakdown of materials for a complex object. Any system of encoding where there is a single outcome from multiple sources like an election or an alert encoding system dependent on multiple trigger points.

The next section of our code declares some of the standard features for our diagram such as the size and shape of the svg container with margins included.

```
var margin = {top: 20, right: 120, bottom: 20, left: 120},
        width = 960 - margin.right - margin.left,
        height = 500 - margin.top - margin.bottom;

var i = 0;

var tree = d3.layout.tree()
        .size([height, width]);
```

It also assigns the variable / function `tree` to the d3.js function[183] that is used to assign and calculate the data required for the nodes and links for our diagram. We will be calling that later.

The next block of code declares the function that will be used to draw the links between the nodes. This isn't the part of the code where the links are drawn, this is just declaring the variable/function that will be used when it does happen.

```
var diagonal = d3.svg.diagonal()
        .projection(function(d) { return [d.y, d.x]; });
```

This uses the d3.js diagonal[184] function to help draw a path between two points such that the line exhibits some nice flowing curves (cubic BÃ©zier) to make the connection.

The next block of code appends our SVG working area to the body of our web page and creates a group elements (`<g>`) that will contain our svg objects (our nodes, text and links).

```
var svg = d3.select("body").append("svg")
        .attr("width", width + margin.right + margin.left)
        .attr("height", height + margin.top + margin.bottom)
  .append("g")
        .attr("transform", "translate(" + margin.left + "," + margin.top + ")");
```

The next line is one that vexed me for a while and one that I think means there are other areas of my code that could be improved (for a short interlude on why this tried me, feel free to catch this question[185] on Stack Overflow).

```
root = treeData[0];
```

It might not look like much and to those familiar with JavaScript, it will be a no-brainer, but what the line is doing is defining what 'tree' from our data is going to be used. Because our data is an array, the first level of the array is `treeData`. The name of the first object on the first level of `treeData` is 'Top Level'. This (being the first object) is object `0`. Therefore our starting point is `treeData[0]`. We could confirm this by changing the declaration to ...

```
root = treeData[0].children[0];
```

This will take the root point for our diagram as being the first child (`child[0]`) of the the first level of `treeData`. As a result, our tree diagram will look like this...

Tree layout diagram using a different root point

... since 'Level 2: A' is the first child of 'Top Level'.

Then we call the function that draws our tree diagram.

[183] https://github.com/mbostock/d3/wiki/Tree-Layout
[184] https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-diagonal
[185] http://stackoverflow.com/questions/20940854/how-to-load-data-from-an-internal-json-array-rather-than-from-an-external-resour

```
update(root);
```

This calls the function `update` and uses the data `root` to create our tree.

The last significant part of the code is the function `update`. This is the part of the code that pulls together the functions and data that we have declared and draws our tree.

The first step in that process is to assign our nodes and links.

```
var nodes = tree.nodes(root),
        links = tree.links(nodes);
```

This uses our previously declared `tree` function to work its d3.js magic on our data (`root`) and to determine the node details and from the node details we can determine the link details.

If you're wondering how this all works, I'm afraid that I won't be able to help much, but a starting point would be the results that the process produces which is a set of nodes, each of which has a set of characteristics. Those characteristics are; - `.children`: Which is an array of any children that exist for that node. - `.depth`: Which is the depth (described in a few paragraphs time). - `.id`: Which is a unique number identifier for each node. - `.name`: The name we have assigned from our data. - `.parent`: The name of the parent of the node. - `.x` and `.y`: Which are the x and y positions on the screen of the node.

From this node data a set of links joining the nodes is created. Each link consists of a `.source` and `.target`. Each of which is a node.

We then determine the horizontal spacing of the nodes.

```
nodes.forEach(function(d) { d.y = d.depth * 180; });
```

This uses the depth of the node (as determined for each node in the `nodes = tree.nodes(root)` line) to calculate the position on the y axis of the screen.

The `depth` refers to the position in the tree relative to the root node on the left. The following picture shows how the depth relates to the position of the node in the tree.

Depth of nodes on the tree

So by adjusting our 'expansion factor' (currently set to 180) we can adjust the spacing of the nodes. For instance, here is the spacing changed to 80.

Adjusting the depth of the tree

We then declare the variable / function `node` so that when we call it later it will know to select the appropriate object (a node) with the appropriate `.id`.

```
var node = svg.selectAll("g.node")
        .data(nodes, function(d) { return d.id || (d.id = ++i); });
```

The next block of code assigns the variable / function `nodeEnter` to the action of appending a node to a particular position.

```
var nodeEnter = node.enter().append("g")
        .attr("class", "node")
        .attr("transform", function(d) {
                return "translate(" + d.y + "," + d.x + ")"; });
```

Then we get to the piece of code that appends the circle that comprises the node (using `nodeEnter`).

```
nodeEnter.append("circle")
        .attr("r", 10)
        .style("fill", "#fff");
```

(using a radius of 10 pixels and a white fill).

And we add in the text for each node...

```
nodeEnter.append("text")
        .attr("x", function(d) {
                return d.children || d._children ? -13 : 13; })
        .attr("dy", ".35em")
        .attr("text-anchor", function(d) {
                return d.children || d._children ? "end" : "start"; })
        .text(function(d) { return d.name; })
        .style("fill-opacity", 1);
```

This is a neat piece of code that allows the text to be placed on the left side of the node if it has children (`d.children`) or on the right if it has has no children or `d._children`. This is a slightly redundant piece of code (the `d._children` piece) for this diagram, but it becomes more useful in the interactive version towards the end of the chapter. It also aligns the text correctly and makes sure it is visible.

Then we declare the link variable / function and tell it to make a link based on all the links that have unique target id's.

```
var link = svg.selectAll("path.link")
        .data(links, function(d) { return d.target.id; });
```

This might not be obvious at first glance, but we only want to draw links between a node and it's parent. There should be one less link than the total number of nodes since the root node ('Top Level') has no parent. Therefore only those links with unique target id's in the data need to have

links produced. If we were to replace the `.target` in the above code with `.source` we would have only two unique `.source` id's. It would therefore look like this;

Only Links from unique sources

Our final block of JavaScript adds in our link as a diagonal path (as declared early in the JavaScript portion of the code).

```
link.enter().insert("path", "g")
        .attr("class", "link")
        .attr("d", diagonal);
```

There are only a couple of lines of HTML to close off the file and we are left with our tree diagram!

Simple tree layout diagram

Don't forget, the full code for this example can be found on github[186], in the appendices of this book or in the code samples bundled with this book (simple-tree-diagram.html). A working example can be found on bl.ocks.org[187].

[186] https://gist.github.com/d3noob/8323795
[187] http://bl.ocks.org/d3noob/8323795

Styling nodes in a tree diagram

The nodes in a tree diagram are objects that exist to provide a representation of the structure of data, but on a tree diagram they should also be viewed as an opportunity to encode additional information about the underlying data.

From the initial simple example that we covered at the start of the chapter we have encoded a certain amount of information already. The position of the text relative to each node is determined by whether or not the node is the parent of another node (if it's a parent it's on the left) or a child that is on the edge of the tree (in which case it is on the right of the node).

Node position on the tree diagram

Now, that's nice, but are we going to be satisfied with that??? (The answer is "No" by the way.)

This example is fairly simple, but it is an example of applying different styles to the nodes to convey additional information. I should be clear at this stage that I am not advocating turning your tree diagram into something that looks like it came out of a circus, because that would be a crime against style, so don't repeat my upcoming example, but let some of the features be a trigger for developing your own subtle, yet compelling visualizations.

Brace yourself. Here's a picture of the tree diagram that we're going to generate. Those with weaker constitutions should look away and flip forward a few pages;

Tree diagram with excessive styling

The changes that have been made are as a result of additional data fields that have been added to the JSON array and these fields have been applied to various style options throughout the code.

The types of style changes we have made are - Variation of the diameter of nodes - Changing the fill and stroke colour of nodes - Changing the colour of links depending on the associated target node they connect to.

We'll start by looking at the new JSON data set;

```
{
  "name": "Top Level",
  "parent": "null",
  "value": 10,
  "type": "black",
  "level": "red",
  "children": [
    {
      "name": "Level 2: A",
      "parent": "Top Level",
```

```
        "value": 15,
        "type": "grey",
        "level": "red",
        "children": [
          {
            "name": "Son of A",
            "parent": "Level 2: A",
            "value": 5,
            "type": "steelblue",
            "level": "orange"
          },
          {
            "name": "Daughter of A",
            "parent": "Level 2: A",
            "value": 8,
            "type": "steelblue",
            "level": "red"
          }
        ]
      },
      {
        "name": "Level 2: B",
        "parent": "Top Level",
        "value": 10,
        "type": "grey",
        "level": "green"
      }
    ]
  }
```

Each node now has a `value` which might represent a degree of importance (we will use this to affect the radius of the nodes), a `type` which might indicate a difference in the type of node (they might be in active, inactive or undetermined states) and a `level` which might indicate an alert level for determining problems (red = bad, orange = caution and green = normal).

Irrespective of the contrived nature of our styling options, they are applied to our tree in fairly similar ways with some subtle differences.

Remember, the full code for this example can be found on github[188] or in the code samples bundled with this book (simple-tree-features.html). A working example can be found on bl.ocks.org[189].

The first change is to the node radius, stroke colour and fill colour.

We simply change the portion of the code that appends the circle from this...

[188] https://gist.github.com/d3noob/8324872
[189] http://bl.ocks.org/d3noob/8324872

```
nodeEnter.append("circle")
        .attr("r", 10)
        .style("fill", "#fff");
```

... to this ...

```
nodeEnter.append("circle")
        .attr("r", function(d) { return d.value; })
        .style("stroke", function(d) { return d.type; })
        .style("fill", function(d) { return d.level; });
```

The changes return the radius attribute as a function using `value`, the stroke colour is returned using `type` and the fill colour is returned with `level`. This is nice and simple, but we do need to make a slight adjustment to the code that sets the distance that the text is from the nodes so that when the radius expands or contracts, the text distance from the edge of the node adjusts as well.

To do this we take the clever piece of code that adjusts the distance that the text is in the x dimension from the node that looks like this ...

```
        .attr("x", function(d) {
                return d.children || d._children ? -13 : 13; })
```

... and we add in a dynamic aspect using the `value` field.

```
        .attr("x", function(d) {
                return d.children || d._children ?
                (d.value + 4) * -1 : d.value + 4 })
```

The last thing we wanted to do is to change the colour of the link based on the colour of the target node. We accomplish this by taking the code that inserts the links...

```
link.enter().insert("path", "g")
        .attr("class", "link")
        .attr("d", diagonal);
```

... and adding in a line that styles the link colour (the stroke) based on the `level` colour of the `target` end of the link `d.target.level`).

```
link.enter().insert("path", "g")
        .attr("class", "link")
        .style("stroke", function(d) { return d.target.level; })
        .attr("d", diagonal);
```

Use the concepts here wisely. I don't want to see any heinously styled tree diagrams floating around the internet with "Thanks to the help from D3 Tips and Tricks" next to them. Be subtle, be thoughtful :-).

Changing the nodes to different shapes

Many thanks to Josiah who asked a question[190] on the d3noob.org blog on how the shapes of the nodes could be varied based on an associated value in the data.

There is more than one way to do this, but perhaps the simplest is to replace the section of the JavaScript that appends the circle with one that appends a symbol from d3's symbol generator[191].

There are six pre-defined symbol types as follows;

- circle - a circle.
- cross - a Greek cross or plus sign.
- diamond - a rhombus.
- square - an axis-aligned square.
- triangle-down - a downward-pointing equilateral triangle.
- triangle-up - an upward-pointing equilateral triangle.

The following script will look at the `value` in the data and assign either a cross or a diamond depending on the `value`

```
nodeEnter.append("path")
  .style("stroke", "black")
  .style("fill", "white")
  .attr("d", d3.svg.symbol()
              .size(200)
              .type(function(d) { if
                 (d.value >= 9) { return "cross"; } else if
                 (d.value <= 9) { return "diamond";}
               }));
```

Tree diagram different node shapes

The full code for this example can be found on github[192] or in the code samples bundled with this book (simple-tree-shapes.html). A working online example can be found on bl.ocks.org[193].

[190] http://www.d3noob.org/2014/01/tree-diagrams-in-d3js_11.html?showComment=1398062145979#c8289012474475643167
[191] https://github.com/mbostock/d3/wiki/SVG-Shapes#symbol
[192] https://gist.github.com/d3noob/11137963
[193] http://bl.ocks.org/d3noob/11137963

Using images as nodes

Many thanks to nbhatta who asked a question[194] on the d3noob.org blog on how to use images as nodes.

Tree diagram with images for nodes

This was a slightly simpler change and just involved replacing the code snippet that added the circles with one that added an image;

```
nodeEnter.append("image")
    .attr("xlink:href", function(d) { return d.icon; })
    .attr("x", "-12px")
    .attr("y", "-12px")
    .attr("width", "24px")
    .attr("height", "24px");
```

The images I chose were all 48 x 48 pixel for the sake of consistency and in the code above I formatted them to be half that size and moved them in the x and y direction so that they were centred correctly.

The cool thing that you will notice is that the specific icon that is placed at each node position is set by the name of the icon which is gathered from the JSON file with the tree details;

```
var treeData = [
  {
    "name": "Top Level",
    "parent": "null",
    "value": 10,
    "type": "black",
    "level": "red",
    "icon": "earth.png",
    "children": [
      {
        "name": "Level 2: A",
        "parent": "Top Level",
        "value": 5,
        "type": "grey",
        "level": "red",
       "icon": "cart.png",
        "children": [
          {
            "name": "Son of A",
            "parent": "Level 2: A",
```

[194] http://www.d3noob.org/2014/01/tree-diagrams-in-d3js_11.html?showComment=1399025834455#c4911480988339223702

```
            "value": 5,
            "type": "steelblue",
            "icon": "lettern.png",
            "level": "orange"
          },
          {
            "name": "Daughter of A",
            "parent": "Level 2: A",
            "value": 18,
            "type": "steelblue",
            "icon": "vlc.png",
            "level": "red"
          }
        ]
      },
      {
        "name": "Level 2: B",
        "parent": "Top Level",
        "value": 10,
        "type": "grey",
        "icon": "random.png",
        "level": "green"
      }
    ]
  }
];
```

It's possible to just have a single image and to hard-code it into the script, but where's the fun in that?

The full code for this example can be found on github[195] or in the code samples bundled with this book (simple-tree-images.html, cart.png, earth.png, lettern.png, random.png and vlc.png). A working online example can be found on bl.ocks.org[196].

[195] https://gist.github.com/d3noob/9662ab6d5ac823c0e444
[196] http://bl.ocks.org/d3noob/9662ab6d5ac823c0e444

Making a vertical tree diagram

Changing a tree diagram from a horizontal view to a vertical one is fairly easy. There are only three things to change from the code that we used for our original simple tree diagram.

The first is to change the orientation of the nodes by transposing the x and y coordinates.

That means taking the section of code that appends the nodes...

```
var nodeEnter = node.enter().append("g")
        .attr("class", "node")
        .attr("transform", function(d) {
                return "translate(" + d.y + "," + d.x + ")"; });
```

... and swapping the `d.x` and `d.y` designators so that it looks like this...

```
var nodeEnter = node.enter().append("g")
        .attr("class", "node")
        .attr("transform", function(d) {
                return "translate(" + d.x + "," + d.y + ")"; });
```

Because the vertical version of the tree diagram can be a lot more compact, we can adjust our difference between the depths to a more rational value. In our example we can change the separation from 180 to 100 pixels in the following line of code...

```
nodes.forEach(function(d) { d.y = d.depth * 100; });
```

The second is to do the same adjustment for the links. We take the block of code that generates the curvy diagonal paths...

```
var diagonal = d3.svg.diagonal()
        .projection(function(d) { return [d.y, d.x]; });
```

... and swap the `d.x` and `d.y` designators so that it looks like this...

```
var diagonal = d3.svg.diagonal()
        .projection(function(d) { return [d.x, d.y]; });
```

At this point we have our tree diagram ready to go except for one small detail...

Vertical tree diagram with sideways text

The text is still aligned to the left and right of the nodes. On this example, it looks pretty good, but if we were to introduce a few more nodes, it would start to get pretty cramped, so we can place the text above and below the nodes dependent on whether the node is a parent (above) or a child on the bottom level (below).

To do this we take the original text appending code...

```
nodeEnter.append("text")
        .attr("x", function(d) {
                return d.children || d._children ? -13 : 13; })
        .attr("dy", ".35em")
        .attr("text-anchor", function(d) {
                return d.children || d._children ? "end" : "start"; })
        .text(function(d) { return d.name; })
        .style("fill-opacity", 1);
```

... and change the `x` attribute to a `y` attribute, anchor the text in the middle (which is actually a simplification of the code) and extend the distance between the node and the anchor point slightly to 18 (and -18) pixels.

```
nodeEnter.append("text")
        .attr("y", function(d) {
                return d.children || d._children ? -18 : 18; })
        .attr("dy", ".35em")
        .attr("text-anchor", "middle")
        .text(function(d) { return d.name; })
        .style("fill-opacity", 1);
```

And there we have it! A vertical tree diagram.

Vertical tree diagram

The full code for this example can be found on github[197] or in the code samples bundled with this book (simple-tree-vertical.html). A working online example can be found on bl.ocks.org[198].

[197] https://gist.github.com/d3noob/8326869
[198] http://bl.ocks.org/d3noob/8326869

Generating a tree diagram from 'flat' data

Tree diagrams are a fantastic way of displaying information, but one of the drawbacks (to the examples we've been using so far) is the need to have your data encoded hierarchically. Most data in a raw form will be flat. That is to say, it won't be formatted as an array with the parent - child relationships. Instead it will be a list of objects (which we will want to turn into nodes) that might describe the relationship to each other, but they won't be encoded that way. For example, the following is the flat representation of the example data we have been using thus far.

```
{ "name" : "Level 2: A", "parent":"Top Level" },
{ "name" : "Top Level", "parent":"null" },
{ "name" : "Son of A", "parent":"Level 2: A" },
{ "name" : "Daughter of A", "parent":"Level 2: A" },
{ "name" : "Level 2: B", "parent":"Top Level" }
```

It is actually fairly simple and consists of only the name of the node and the name of it's parent node. It's easy to see how this data could be developed into a hierarchical form, but it would take a little time and for a larger data set, that would be tiresome.

Luckily computers are built for shuffling data about and with kudos to 'nrabinowitz' for answering a question[199] (and Prateek Tandon for asking) on Stack Overflow (and Jesus Ruiz with AmeliaBR for setting me on the right path[200]), here is how we can take our flat data and convert it for use in our tree diagram.

We will be using the simple example that we started with at the start of the chapter and the first change we need to make is to replace our original data…

```
var treeData = [
  {
    "name": "Top Level",
    "parent": "null",
    "children": [
      {
        "name": "Level 2: A",
        "parent": "Top Level",
        "children": [
          {
            "name": "Son of A",
            "parent": "Level 2: A"
          },
          {
            "name": "Daughter of A",
            "parent": "Level 2: A"
          }
```

[199] http://stackoverflow.com/questions/17847131/generate-multilevel-flare-json-data-format-from-flat-json
[200] http://stackoverflow.com/questions/20940854/how-to-load-data-from-an-internal-json-array-rather-than-from-an-external-resour

```
        ]
      },
      {
        "name": "Level 2: B",
        "parent": "Top Level"
      }
    ]
  }
];
```

... with our flat data array...

```
var data = [
    { "name" : "Level 2: A", "parent":"Top Level" },
    { "name" : "Top Level", "parent":"null" },
    { "name" : "Son of A", "parent":"Level 2: A" },
    { "name" : "Daughter of A", "parent":"Level 2: A" },
    { "name" : "Level 2: B", "parent":"Top Level" }
    ];
```

It's worth noting here that we have also changed the name of the array (to `data`) since we are going to convert, then declare our newly massaged data with our original variable name `treeData` so that the remainder of our code thinks there have been no changes.

Then we create a name-based map for the nodes. In his answer on Stack Overflow, 'nrabinowitz' uses the .reduce method[201], which starts with an empty object and iterates over the data array, adding an entry for each node.

```
var dataMap = data.reduce(function(map, node) {
        map[node.name] = node;
        return map;
}, {});
```

Don't feel upset if you don't understand exactly how it works. I struggle to understand internal combustion engines, but I'm ok at driving a car :-). Think of this in the same way.

Then we iteratively add each child to its parents, or to the root array if no parent is found;

[201] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce

```
var treeData = [];
data.forEach(function(node) {
        // add to parent
        var parent = dataMap[node.parent];
        if (parent) {
                // create child array if it doesn't exist
                (parent.children || (parent.children = []))
                        // add node to child array
                        .push(node);
        } else {
                // parent is null or missing
                treeData.push(node);
        }
});
```

The code is essentially working through each node in the array and if it has a child it adds it to the `children` sub-array and if necessary creates the array. Likewise, if the node has no parent, it simply add it as a root node.

That's it!

The brevity of the code to do this should not detract from its elegance. It really is very clever. The end result doesn't look any different from our original diagram…

Tree diagram (but from flat data)

… but it adds a significant capability for use of additional data.

The full code for this example can be found on github[202] or in the code samples bundled with this book (simple-tree-from-flat.html). A working example can be found on bl.ocks.org[203].

[202] https://gist.github.com/d3noob/8329404
[203] http://bl.ocks.org/d3noob/8329404

Generating a tree diagram from external data

In all the examples we have looked at so far we have used data that we have declared from within the file itself. Being able to import data from an external file is an important feature that we need to know how to implement.

Starting from the simple tree diagram example that we began with at the start of the chapter, the first change that we need to make is to remove the section of code that declares our data. But don't throw it away since we will use it to create a separate file called `treeData.json`. It's contents will be;

```
[
  {
    "name": "Top Level",
    "parent": "null",
    "children": [
      {
        "name": "Level 2: A",
        "parent": "Top Level",
        "children": [
          {
            "name": "Son of A",
            "parent": "Level 2: A"
          },
          {
            "name": "Daughter of A",
            "parent": "Level 2: A"
          }
        ]
      },
      {
        "name": "Level 2: B",
        "parent": "Top Level"
      }
    ]
  }
]
```

(don't include the `treeData =` part, or the semicolon at the end (you *can* delete those))

Then all we need to do is change the portion of the code that declared the `root` variable and updates the diagram;

```
root = treeData[0];

update(root);
```

... into a small section that uses the `d3.json` accessor to load the file `treeData.json` (Remember to correctly address the file. This one assumes that the `treeData.json` file is in the same directory as the html file we are opening).

```
d3.json("treeData.json", function(error, treeData) {
  root = treeData[0];
  update(root);
});
```

It then declares the variable root in the same way and calls the update function to draw the tree diagram. Viola!

The full code for this example can be found on github[204] or in the code samples bundled with this book (simple-tree-from-external.html and treeData.json). A working example can be found on bl.ocks.org[205].

[204] https://gist.github.com/d3noob/8329447
[205] http://bl.ocks.org/d3noob/8329447

Generating a tree diagram from a CSV file.

Creating a tree diagram from a csv file is an extension of the sections where we create a diagram from flat data and where we create a diagram from an external file.

By mashing these together and using a csv file something like the following…

```
name,parent
Level 2: A,Top Level
Top Level,null
Son of A,Level 2: A
Daughter of A,Level 2: A
Level 2: B,Top Level
```

… we can ingest the name of the nodes and their relationships and then format the data correctly.

Tree diagram (but from csv)

The full code for this example can be found on github[206] or in the code samples bundled with this book (simple-tree-from-csv.html and treedata.csv). A working example can be found on bl.ocks.org[207].

[206] https://gist.github.com/d3noob/fa0f16e271cb191ae85f
[207] http://bl.ocks.org/d3noob/fa0f16e271cb191ae85f

An interactive tree diagram

The examples presented thus far have all been static in the sense that they present information on a web page, but that's where they stop. One of the strengths of web content is the ability to involve the reader to a greater extent. Therefore the following tree diagram example includes an interactive element where the user can click on any parent node and it will collapse on itself to make more room for others or to simplify a view. Additionally, any collapsed parent node can be clicked on and it will re-grow to its previous condition.

The example included here is a close derivative of Mike Bostock's example[208]. I won't fully explain the operation of this file, but we will consider parts of it for interest's sake.

The full code for this example can be found on github[209], in the appendices of this book or in the code samples bundled with this book (interactive-tree.html). A working online example can be found on bl.ocks.org[210].

For a brief visual description of the action. The diagram will initially display the complete tree…

Tree diagram)

Then when clicking on the 'Level 2: A' node, the tree partially collapses to…

Partially collapsed tree diagram)

We could also click on the root node ('Top Level') to fully collapse the tree…

Fully collapsed tree diagram)

Then clicking on the nodes opens the diagram back up again.

One of the important changes to start with is to make each node responsive to the mouse pointer. This is done by including the following in the `<style>` section.

```
.node {
        cursor: pointer;
}
```

The code then adds sections to allow the diagram to follow the d3.js model of enter - update - exit for the nodes with a suitable transition in between.

[208] http://bl.ocks.org/mbostock/4339083
[209] https://gist.github.com/d3noob/8375092
[210] http://bl.ocks.org/d3noob/8375092

Nodes are coloured ("steelblue") if they have been collapsed and at the end of the script we have a function that makes use of the `d._children` reference we have been using in most of our examples.

```
function click(d) {
  if (d.children) {
        d._children = d.children;
        d.children = null;
  } else {
        d.children = d._children;
        d._children = null;
  }
  update(d);
}
```

This allows the action of clicking on the nodes to update the data associated with the node and as a consequence change it's properties in the script based on if statements (Such as `"fill", function(d) { return d._children ? "lightsteelblue" : "#fff"; }` which will fill the node with "lightsteelblue" if `d._children` exists, otherwise make it white.)

The examples we have looked at in the previous sections in this chapter are all applicable to this interactive version, so this should provide you with the capability to generate some interesting visualizations.

Enjoy.

Force Layout Diagrams

What is a Force Layout Diagram?

This is not a distinct type of diagram per se. Instead, it's a way of representing data so that individual data points share relationships to other data points via forces. Those forces can then act in different ways to provide a natural structure to the data. The end result can be a wide variety of representations of connectedness and groupings.

Mike Bostock gave a great talk which focussed on force layout techniques in 2011 at Trulia for the Data Visualization meetup group. Check video of the presentation here: http://vimeo.com/29458354[211] and the slides here: http://mbostock.github.com/d3/talk/20110921/#0[212]. The most memorable quote I recall from the talk describes force layout diagrams as an "*Implicit way to do position encoding*".

Here's some examples for those who need a reason to view the talk.

Multi-Foci Force Layout

Multi-Foci Force Layout

Simultaneous forces of repulsion and multiple gravitational focus points create a natural clustering of data points (Source: Mike Bostock http://bl.ocks.org/mbostock/1249681[213]). The graph is animated, so the artefacts such as overlapping circles and the purple circle that is located beside the red area are transitory.

Force Directed Graph with Pan / Zoom

[211] http://vimeo.com/29458354
[212] http://mbostock.github.com/d3/talk/20110921/#0
[213] http://bl.ocks.org/mbostock/1249681

Force Directed Graph with Pan / Zoom

Multiple linked nodes show connections between related entities where those entities are labelled and encoded with relevant information. Created by David Graus and presented here: http://graus.nu/blog/force-directed-graphs-playing-around-with-d3-js/[214].

Collapsible Force Layout

Collapsible Force Layout

This force directed graph can have individual nodes expanded or collapsed by clicking on them to reveal or hide greater detail (Source: Mike Bostock http://bl.ocks.org/mbostock/1062288[215]).

Force Directed Graph showing Directionality

[214] http://graus.nu/blog/force-directed-graphs-playing-around-with-d3-js/
[215] http://bl.ocks.org/mbostock/1062288

Force Directed Graph showing Directionality

This example showing mobile patent lawsuits between companies presents the direction associated with the links and encodes the links to show different types (Source: Mike Bostock http://bl.ocks.org/mbostock/1153292[216]).

Collision Detection

Collision Detection

In this example the mouse exerts a repulsive force on the objects as it moves on the screen (Source: Mike Bostock http://bl.ocks.org/mbostock/3231298[217]).

Molecule Diagram

[216] http://bl.ocks.org/mbostock/1153292
[217] http://bl.ocks.org/mbostock/3231298

Molecule Diagram

Just for fun, here is a diagram that Mike Bostock made to demonstrate drawing two parallel lines between nodes. He's the first to admit that increasing the number of lines becomes awkward, but it serves as another example of the flexibility of force diagrams in D3 (Source: Mike Bostock http://bl.ocks.org/mbostock/3037015[218]).

The main forces in play in these diagrams are charge, gravity and friction. More detailed information on these forces and the other parameters associated with the force layout code can be found in the D3 Wiki[219].

Charge

Charge is a force that a node can exhibit where it can either attract (positive values) or repel (negative values). Varying this value in conjunction with other forces (such as gravity) or a link (on a node by node basis) is generally necessary to maintain stability.

Gravity

The gravity force isn't actually a true representation of gravitational attraction (this can be more closely approximated using positive values of charge). Instead it approximates the action of a spring connected to a node. This has a more pleasant visual effect when the affected node is closer to its 'great attractor' and avoids what would otherwise be a small black hole type effect.

Friction

The frictional force is one designed to act on the movement of a node to reduce its speed over time. It isn't implemented as true friction (in the physical sense) and should be thought of as a 'velocity decay' in the truer sense.

Mike makes the point in the 2011 talk at Trulia that when using gravity in a force layout diagram, it is useful to include a degree of charge repulsion to provide stability. This can be demonstrated by experimenting with varying values of the charges in a diagram and observing the effects.

Force directed graph examples.

There are a large number of possible examples to use to demonstrate force directed graphs. I chose to combine two examples that Mike Bostock has demonstrated in the past. Both use the

[218] http://bl.ocks.org/mbostock/3037015
[219] https://github.com/mbostock/d3/wiki/Force-Layout

data for the 'who's suing who' graph because I wanted especially to include the directionality aspect of the links. The two graphs I based the final graph on were the Mobile Patent Suits[220] graph....

Mobile Patent Suits

... for the directionality and link encoding and the Force-Directed Graph with Mouseover[221] graph...

Force-Directed Graph with Mouseover

... for the mouseover effects (note the enlarged 'Microsoft' circle).

In spite of the similarities to each other in terms of data and network linkages, the final example code was quite different, so the end result is a distinct hybrid of the two and will look something like this;

[220] http://bl.ocks.org/mbostock/1153292
[221] http://bl.ocks.org/mbostock/2706022

Force-Directed Graph with Node Highlighting and Link Value Gradients

In this example the nodes can be clicked on once to enlarge the associated circle and text and then double clicked on to return them to normal. The links vary in opacity depending on an associated value loaded with the data. The example code for the graph above will be explained later in the chapter and can be found on bl.ocks.org[222] or in the code samples bundled with this book (force-highlight-opacity.html and force.csv).

Basic force directed graph showing directionality

The data for this graph has been altered from the data that was comprised of litigants in the mobile patent war to fictitious people's names and associated values (to represent the strength of the links between the two).

The full code for this diagram can also be found on github[223] or in the code samples bundled with this book (force.html and force.csv). A live example can be found on bl.ocks.org[224].

In the original examples the data was contained in the graph code. In the following example it is loaded from a csv file. The values loaded are as follows;

```
source,target,value
Harry,Sally,1.2
Harry,Mario,1.3
Sarah,Alice,0.2
Eveie,Alice,0.5
Peter,Alice,1.6
Mario,Alice,0.4
James,Alice,0.6
Harry,Carol,0.7
Harry,Nicky,0.8
Bobby,Frank,0.8
Alice,Mario,0.7
Harry,Lynne,0.5
Sarah,James,1.9
Roger,James,1.1
```

[222] http://bl.ocks.org/d3noob/5155181
[223] https://gist.github.com/d3noob/5141278
[224] http://bl.ocks.org/d3noob/5141278

```
Maddy,James,0.3
Sonny,Roger,0.5
James,Roger,1.5
Alice,Peter,1.1
Johan,Peter,1.6
Alice,Eveie,0.5
Harry,Eveie,0.1
Eveie,Harry,2.0
Henry,Mikey,0.4
Elric,Mikey,0.6
James,Sarah,1.5
Alice,Sarah,0.6
James,Maddy,0.5
Peter,Johan,0.7
```

The code is as follows;

```
<!DOCTYPE html>
<meta charset="utf-8">
<script src="http://d3js.org/d3.v3.js"></script>

<style>


path.link {
  fill: none;
  stroke: #666;
  stroke-width: 1.5px;
}

circle {
  fill: #ccc;
  stroke: #fff;
  stroke-width: 1.5px;
}

text {
  fill: #000;
  font: 10px sans-serif;
  pointer-events: none;
}
```

```
</style>
<body>
<script>


// get the data
d3.csv("force.csv", function(error, links) {

var nodes = {};

// Compute the distinct nodes from the links.
links.forEach(function(link) {
    link.source = nodes[link.source] ||
        (nodes[link.source] = {name: link.source});
    link.target = nodes[link.target] ||
        (nodes[link.target] = {name: link.target});
    link.value = +link.value;
});

var width = 960,
    height = 500;

var force = d3.layout.force()
    .nodes(d3.values(nodes))
    .links(links)
    .size([width, height])
    .linkDistance(60)
    .charge(-300)
    .on("tick", tick)
    .start();

var svg = d3.select("body").append("svg")
    .attr("width", width)
    .attr("height", height);

// build the arrow.
svg.append("svg:defs").selectAll("marker")
    .data(["end"])      // Different link/path types can be defined here
  .enter().append("svg:marker")    // This section adds in the arrows
    .attr("id", String)
    .attr("viewBox", "0 -5 10 10")
    .attr("refX", 15)
    .attr("refY", -1.5)
    .attr("markerWidth", 6)
    .attr("markerHeight", 6)
    .attr("orient", "auto")
  .append("svg:path")
```

```
    .attr("d", "M0,-5L10,0L0,5");

// add the links and the arrows
var path = svg.append("svg:g").selectAll("path")
    .data(force.links())
  .enter().append("svg:path")
    .attr("class", "link")
    .attr("marker-end", "url(#end)");

// define the nodes
var node = svg.selectAll(".node")
    .data(force.nodes())
  .enter().append("g")
    .attr("class", "node")
    .call(force.drag);

// add the nodes
node.append("circle")
    .attr("r", 5);

// add the text
node.append("text")
    .attr("x", 12)
    .attr("dy", ".35em")
    .text(function(d) { return d.name; });

// add the curvy lines
function tick() {
    path.attr("d", function(d) {
        var dx = d.target.x - d.source.x,
            dy = d.target.y - d.source.y,
            dr = Math.sqrt(dx * dx + dy * dy);
        return "M" +
            d.source.x + "," +
            d.source.y + "A" +
            dr + "," + dr + " 0 0,1 " +
            d.target.x + "," +
            d.target.y;
    });

    node
        .attr("transform", function(d) {
                    return "translate(" + d.x + "," + d.y + ")"; });
}

});
```

```
</script>
</body>
</html>
```

In a similar process to the one we went through when highlighting the function of the Sankey diagram, where there are areas that we have covered before, I will gloss over some details on the understanding that you will have already seen them explained in an earlier section (most likely the basic line graph example).

The first block we come across is the initial html section;

```
<!DOCTYPE html>
<meta charset="utf-8">
<script src="http://d3js.org/d3.v3.js"></script>

<style>
```

The only thing slightly different with this example is that we load the d3.v3.js script earlier. This has no effect on running the code.

The next section loads the Cascading Style Sheets;

```
path.link {
  fill: none;
  stroke: #666;
  stroke-width: 1.5px;
}

circle {
  fill: #ccc;
  stroke: #fff;
  stroke-width: 1.5px;
}

text {
  fill: #000;
  font: 10px sans-serif;
  pointer-events: none;
}
```

We set styles for three elements and all the settings laid out are familiar to us from previous work.

Then we move into the JavaScript. Our first line loads our csv data file (`force.csv`).

```
d3.csv("force.csv", function(error, links) {
```

Then we declare an empty object (I still tend to think of these as arrays even though they're strictly not).

```
var nodes = {};
```

This will contain our data for our nodes. We don't have any separate node information in our data file, it's just link information, so we will be populating this in the next section...

```
links.forEach(function(link) {
    link.source = nodes[link.source] ||
        (nodes[link.source] = {name: link.source});
    link.target = nodes[link.target] ||
        (nodes[link.target] = {name: link.target});
    link.value = +link.value;
});
```

This block of code looks through all of our data from our csv file and for each link adds it as a node if it hasn't seen it before. It's quite clever how it works as it employs a neat JavaScript shorthand method using the double pipe (`||`) identifier.

So the line (expanded)...

```
link.source=nodes[link.source] || (nodes[link.source]={name: link.source});
```

... can be thought of as saying "*If* `link.source` *does not equal any of the* `nodes` *values then create a new element in the* `nodes` *object with the name of the* `link.source` *value being considered.*". It could conceivably be written as follows (this is untested);

```
if (link.source != nodes[link.source]) {
    nodes[link.source] = {name: link.source}
};
```

Then the block of code goes on to test the `link.target` value in the same way. Then the `value` variable is converted to a number from a string if necessary (`link.value = +link.value;`).

The next block sets the size of our svg area that we'll be using;

```
var width = 960,
    height = 500;
```

The next section introduces the `force` function.

```
var force = d3.layout.force()
    .nodes(d3.values(nodes))
    .links(links)
    .size([width, height])
    .linkDistance(60)
    .charge(-300)
    .on("tick", tick)
    .start();
```

Full details for this function are found on the D3 Wiki[225], but the following is a rough description of the individual settings.

`var force = d3.layout.force()` makes sure we're using the `force` function.

`.nodes(d3.values(nodes))` sets our layout to the array of `nodes` as returned by the function `d3.values` (https://github.com/mbostock/d3/wiki/Arrays#wiki-d3_values[226]). Put simply, it sets the nodes to the `nodes` we have previously set in our object.

`.links(links)` does for links what `.nodes` did for nodes.

`.size([width, height])` sets the available layout size to our predefined values. If we were using `gravity` as a force in the graph this would also set the gravitational centre. It also sets the initial random position for the elements of our graph.

`.linkDistance(60)` sets the target distance between linked nodes. As the graph begins and moves towards a steady state, the distance between each pair of linked nodes is computed and compared to the target distance; the links are then moved towards or away from each other, so as to converge on the set distance.

Setting this value (and other force values) can be something of a balancing act. For instance, here is the result of setting the `.linkDistance` to 160.

[225] https://github.com/mbostock/d3/wiki/Force-Layout
[226] https://github.com/mbostock/d3/wiki/Arrays#wiki-d3_values

Link distance set to 160

Here the charged nodes are trying to arrange themselves at an appropriate distance, but the length of the links means that their arrangement is not very pretty. Likewise if we change the value to 30 we get the following;

Link distance set to 30

Here the link distance allows for a symmetrical layout, but the distance is too short to be practical.

`.charge(-300)` sets the force between nodes. Negative values of `charge` results in node repulsion, while a positive value results in node attraction. In our example, if we vary the value to 150 we get this result;

Charge set to 150

It's not exactly easy to spot, but the graph feels a little 'lazy'. The nodes don't find their equilibrium easily or at all. Setting the value higher than 300 (for our example) keeps all the nodes nice and spread out, but where there are other separate discrete linked nodes (as there are in our example) they tend to get forced away from the centre of the defined area.

`.on("tick", tick)` runs the animation of the force layout one 'step'. It's this progression of steps that gives the force layout diagram it's fluid movement.

`.start();` Starts the simulation; this method must be called when the layout is first created.

The next block of our code is the standard section that sets up our svg container.

```
var svg = d3.select("body").append("svg")
    .attr("width", width)
    .attr("height", height);
```

The next block of our code is used to create our arrowhead marker. I will be the first to admit that it has entered a realm of svg expertise that I do not have and the amount of extra memory power I would need to accumulate to understand it sufficiently to explain won't be occurring in the near future. Please accept my apologies and as a small token of my regret, accept the following links as an invitation to learn more: http://www.w3.org/TR/SVG/coords.html#ViewBoxAttribute[227] and http://www.w3schools.com/svg/svg_reference.asp?[228]. What is useful to note here is that we define the label for our marker as `end`. We will use this in the next section to reference the marker as an object. This particular section of the code caused me some small amount of angst. The problem being when I attempted to adjust the width of the link lines in conjunction with the `value` set in the data for the link, it would also adjust the stroke-width of the arrowhead marker. Then when I attempted to adjust for the positioning of the arrow on the path, I could never get the maths right. Eventually I decided to stop struggling against it and encode the value of the line in a couple of different ways. One as opacity using discrete boundaries and the other using variable line width, but with the arrowheads a common size. We will cover both those solutions in the coming sections.

[227] http://www.w3.org/TR/SVG/coords.html#ViewBoxAttribute
[228] http://www.w3schools.com/svg/svg_reference.asp?

```
svg.append("svg:defs").selectAll("marker")
    .data(["end"])
  .enter().append("svg:marker")
    .attr("id", String)
    .attr("viewBox", "0 -5 10 10")
    .attr("refX", 15)
    .attr("refY", -1.5)
    .attr("markerWidth", 6)
    .attr("markerHeight", 6)
    .attr("orient", "auto")
  .append("svg:path")
    .attr("d", "M0,-5L10,0L0,5");
```

The `.data(["end"])` line sets our tag for a future part of the script to find this block and draw the marker.

`.attr("refX", 15)` sets the offset of the arrow from the centre of the circle. While it is designated as the `X` offset, because the object is rotating, it doesn't correspond to the x (left and right) axis of the screen. The same is true of the `.attr("refY", -1.5)` line.

The `.attr("markerWidth", 6)` and `.attr("markerHeight", 6)` lines set the bounding box for the marker. So varying these will vary the space available for the marker.

The next block of code adds in our links as paths and uses the `#end` marker to draw the arrowhead on the end of it.

```
var path = svg.append("svg:g").selectAll("path")
    .data(force.links())
  .enter().append("svg:path")
    .attr("class", "link")
    .attr("marker-end", "url(#end)");
```

Then we define what our nodes are going to be.

```
var node = svg.selectAll(".node")
    .data(force.nodes())
  .enter().append("g")
    .attr("class", "node")
    .call(force.drag);
```

This uses the `nodes` data and adds the `.call(force.drag);` function which allows the node to be dragged by the mouse.

The next block adds the nodes as an svg circle.

```
node.append("circle")
    .attr("r", 5);
```

And then we add the name of the node with a suitable offset.

```
node.append("text")
    .attr("x", 12)
    .attr("dy", ".35em")
    .text(function(d) { return d.name; });
```

The last block of JavaScript is the `ticks` function. This block is responsible for updating the graph and most interestingly drawing the curvy lines between nodes.

```
function tick() {
    path.attr("d", function(d) {
        var dx = d.target.x - d.source.x,
            dy = d.target.y - d.source.y,
            dr = Math.sqrt(dx * dx + dy * dy);
        return "M" +
            d.source.x + "," +
            d.source.y + "A" +
            dr + "," + dr + " 0 0,1 " +
            d.target.x + "," +
            d.target.y;
    });

    node
        .attr("transform", function(d) {
                    return "translate(" + d.x + "," + d.y + ")"; });
}
```

This is another example where there are some easily recognisable parts of the code that set the x and y points for the ends of each link (`d.source.x`, `d.source.y` for the start of the curve and `d.target.x`, `d.target.y` for the end of the curve) and a transformation for the node points, but the cleverness is in the combination of the math for the radius of the curve (`dr = Math.sqrt(dx * dx + dy * dy);`) and the formatting of the svg associated with it. This is sadly beyond the scope of what I can comfortable explain, so we will have to be content with "the magic happens here".

The end result should be a tidy graph that demonstrates nodes and directional links between them.

Basic Directional Force Layout Diagram

Directional Force Layout Diagram (Node Highlighting)

Following on from the Basic Force Layout Diagram, our next goal is to highlight our nodes so that we can get a better view of what ones they are (the view can get a little crowded as the nodes begin to increase in number).

To do this we are going to use a couple more of the mouse events that we first introduced in the tooltips section.

For this example we are going to use the `click` event (Triggered by a mouse click (mousedown and then mouseup over an element)) and the `dblclick` event (Triggered by two clicks within a short time over an element).

The single click will enlarge the node and the associated text and the double click will return the node and test to its original size.

The way to implement this is to first set a hook to capture when the event occurs, which calls a function which is laid out later in the script.

The hook is going to be part of the JavaScript where we define our nodes;

```
var node = svg.selectAll(".node")
    .data(force.nodes())
  .enter().append("g")
    .attr("class", "node")
    .on("click", click)        // Add in this line
    .on("dblclick", dblclick)  // Add in this line too
    .call(force.drag);
```

The two additional lines above tell the script that when it sees a click or a double-click on the node (since it's part of the node set-up) to run either the `click` or `dblclick` functions.

The following two function blocks should be placed after the `tick` function but before the closing curly bracket and bracket as indicated;

```
function tick() {
    path.attr("d", function(d) {
        var dx = d.target.x - d.source.x,
            dy = d.target.y - d.source.y,
            dr = Math.sqrt(dx * dx + dy * dy);
        return "M" +
            d.source.x + "," +
            d.source.y + "A" +
            dr + "," + dr + " 0 0,1 " +
            d.target.x + "," +
            d.target.y;
    });

    node
        .attr("transform", function(d) {
                    return "translate(" + d.x + "," + d.y + ")"; });
}

        // <= Put the functions in here!

});
```

The `click` function is as follows;

```
function click() {
    d3.select(this).select("text").transition()
        .duration(750)
        .attr("x", 22)
        .style("fill", "steelblue")
        .style("stroke", "lightsteelblue")
        .style("stroke-width", ".5px")
        .style("font", "20px sans-serif");
    d3.select(this).select("circle").transition()
        .duration(750)
        .attr("r", 16)
        .style("fill", "lightsteelblue");
}
```

The first line declares the function name (`click`). Then we select the node that we've clicked on and then the associated text before we begin the declaration for our transition with the following;

```
d3.select(this).select("text").transition()
```

Then we define the new properties that will be in place after the transition. We move the text's x position (`.attr("x", 22)`), make the text fill steel blue (`.style("fill", "steelblue")`), set the stroke around the edge of the text light steel blue (`.style("stroke", "lightsteelblue")`), set

that stroke to half a pixel wide (`.style("stroke-width", ".5px")`) and increase the font size to 20 pixels (`.style("font", "20px sans-serif");`).

Then we do much the same for the circle component of the node. Select it, declare the transition, increase the radius and change the fill colour.

The `dblclick` function does exactly the same as the `click` function, but reverses the action to return the text and circle to the original settings.

```
function dblclick() {
    d3.select(this).select("circle").transition()
        .duration(750)
        .attr("r", 6)
        .style("fill", "#ccc");
    d3.select(this).select("text").transition()
        .duration(750)
        .attr("x", 12)
        .style("stroke", "none")
        .style("fill", "black")
        .style("stroke", "none")
        .style("font", "10px sans-serif");
}
```

The end result is a force layout diagram where you can click on nodes to increase their size (circle and text) and then double click to reset them if desired.

Directional Force Layout Diagram (Node Highlighting)

The full code for this diagram can be found on github[229] or in the code samples bundled with this book (force-highlight.html and force.csv). A live example can be found on bl.ocks.org[230].

[229] https://gist.github.com/d3noob/5141528
[230] http://bl.ocks.org/d3noob/5141528

Directional Force Layout Diagram (varying link opacity)

The next variation to our force layout diagram is the addition of variation in the link to represent different values (think of the number of packets passed or the amount of money transferred).

There are a few different ways to do this, but by virtue of the inherent close linkages between the arrowhead marker and the link line, altering both in synchronicity proved to be beyond my meagre talents. However, I did find a couple of suitable alternatives and I will go through one here.

In this example we will take the value associated in the loaded data with the link and we will adjust the opacity of the link line in a staged way according to the range of values.

For example, in a range of link strengths from 0 to 100, the bottom 25% will be at opacity 0.25, from 25 to 50 will be 0.50, 50 to 75 will be 0.75 and above 75 will have an opacity of 1. So the final result looks a little like this;

Directional Force Layout Diagram (varying link opacity)

The changes to the code to create this effect are focussed on creating an appropriate range for the values associated with the links and then applying the opacity according to that range in discrete steps.

The first change to the node highlighting code that we make is to the style section. The following elements are added;

```
path.link.twofive {
  opacity: 0.25;
}

path.link.fivezero {
  opacity: 0.50;
}

path.link.sevenfive {
  opacity: 0.75;
}

path.link.onezerozero {
```

```
  opacity: 1.0;
}
```

This provides our four different 'classes' of opacity.

Then in a block of code that comes just after the declaration of the `force` properties we have the following;

```
var        v = d3.scale.linear().range([0, 100]);

v.domain([0, d3.max(links, function(d) { return d.value; })]);

links.forEach(function(link) {
        if (v(link.value) <= 25) {
                link.type = "twofive";
        } else if (v(link.value) <= 50 && v(link.value) > 25) {
                link.type = "fivezero";
        } else if (v(link.value) <= 75 && v(link.value) > 50) {
                link.type = "sevenfive";
        } else if (v(link.value) <= 100 && v(link.value) > 75) {
                link.type = "onezerozero";
        }
});
```

Here we set the scale and the range for the variable `v` (`var v = d3.scale.linear().range([0, 100]);`). We then set the domain for v to go from `0` to the maximum `value` that we have in our link data.

The final block above uses a cascading set of if statements to assign a label to the `type` parameter of each link. This label is the linkage back to the styles we defined previously.

The final change is to take the line where we assigned a class of `link` to each link previously...

```
    .attr("class", "link")
```

...and add in our `type` parameter as well;

```
    .attr("class", function(d) { return "link " + d.type; })
```

Obviously if we wanted a greater number of opacity levels we would add in further style blocks (with the appropriate values) and modify our cascading if statements. I'm not convinced that this solution is very elegant for what I'm trying to do (it was a much better fit for the application that Mike Bostock applied it to originally where he designated different types of law suits) but I'll take the result as a suitable way of demonstrating variation of value.

The full code for this diagram can be found on github[231] or in the code samples bundled with this book (force-highlight-opacity.html and force.csv). A live example can be found on bl.ocks.org[232].

The full code for the Directional Force Layout Diagram with varying link opacity is also in the Appendix: Force Layout Diagram at the rear of the book.

[231] https://gist.github.com/d3noob/5155181
[232] http://bl.ocks.org/d3noob/5155181

Directional Force Layout Diagram (Unique Node Colour)

The following example was put together in response to a question on the d3noob.org[233] site from 'Gino'. While the example isn't precisely what Gino was wanting to achieve, it does illustrate the application of a colour palette to unique elements.

The end result looks like the following;

Directional Force Layout Diagram (Unique Node Colour)

Here each of the nodes has had a separate colour applied to it from one of the 20 colour palette categorical colour ranges. An excellent overview of these ranges is on the d3 wiki[234].

The full code for this diagram can be found on github[235] or in the code samples bundled with this book (force-colour-nodes.html and force.csv). A live example can be found on bl.ocks.org[236].

The changes required from the previous example with the altered opacity are pretty simple.

Firstly we declare the colour range we're going to use in the variable section.

```
color = d3.scale.category20c();
```

In this case we'll use the `category20c` range.

Then we add the fill style for the circle to the code where we append the circles to our graphic.

```
node.append("circle")
    .attr("r", 5)
    .style("fill", function(d) { return color(d.name); });
```

The code applies the fill based on a function that returns a different colour based on each unique node name. So just to be clear here. We're not setting a specific colour to a node. The colours are assigned as a function of where each name sits in the array of nodes (practically random, but in an ordered way :-)).

Then remove the style declarations in the `function click()` and `function dblclick()` where the fill colour is declared for the circles. This prevents the colours from turning grey or steelblue when they are clicked / double clicked. This means that we can click on a few of our new coloured nodes and their unique colours are retained thusly...

Directional Force Layout Diagram ('clicked' unique node colours)

Good question Gino. Many thanks.

[233] http://www.d3noob.org/2013/03/d3js-force-directed-graph-example-basic.html?showComment=1387379478999#c289010472197093679
[234] https://github.com/mbostock/d3/wiki/Ordinal-Scales#categorical-colors
[235] https://gist.github.com/d3noob/8043434
[236] http://bl.ocks.org/d3noob/8043434

Bullet Charts

Introduction to bullet chart structure

One of the first D3.js examples I ever came across (back when Protovis was the thing to use) was one with bullet charts (or bullet graphs).

It struck me straight away as an elegant way to represent data by providing direct information and context.

Bullet Chart

The Bullet Graph Design Specification[237] was laid down by Stephen Frew as part of his work with Perceptual Edge[238].

Using his specification we can break down the components of the chart as follows.

Bullet Chart Specification

Text label: Identifies the performance measure being represented.

Quantitative scale: A scale that is an analogue of the scale on the x axis of a two dimensional xy graph.

Performance measure: The primary data being displayed. In this case the frequency of operation of a CPU.

Comparative marker: A reference symbol designating a measurement such as the previous day's high value (or similar).

[237] http://www.perceptualedge.com/articles/misc/Bullet_Graph_Design_Spec.pdf
[238] http://www.perceptualedge.com/

Qualitative ranges: These represent ranges such as low, medium and high or bad, satisfactory and good. Ideally there would be no fewer than two and no more than 5 of these (for the purposes of readability).

Understanding the specification for the chart is useful, because it's also reflected in the way that the data for the chart is structured.

For instance, If we take the current example, the data can be presented (in JSON) as follows;

```
[
  {
    "title":"CPU 1 Load",
    "subtitle":"GHz",
    "ranges":[1500,2250,3000],
    "measures":[2200],
    "markers":[2500]
  }
]
```

Here we an see all the components for the chart laid out and it's these values that we will load into our D3 script to display.

D3.js code for bullet charts

We'll move through the explanation of the code in a similar process to the other examples in the book. Where there are areas that we have covered before, I will gloss over some details on the understanding that you will have already seen them explained in an earlier section (most likely the basic line graph example).

Here is the full code;

```
<!DOCTYPE html>
<meta charset="utf-8">
<style>


body {
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
  margin: auto;
  padding-top: 40px;
  position: relative;
  width: 800px;
}

button {
  position: absolute;
  right: 40px;
```

```
  top: 10px;
}

.bullet { font: 10px sans-serif; }
.bullet .marker { stroke: #000; stroke-width: 2px; }
.bullet .tick line { stroke: #666; stroke-width: .5px; }
.bullet .range.s0 { fill: #eee; }
.bullet .range.s1 { fill: #ddd; }
.bullet .range.s2 { fill: #ccc; }
.bullet .measure.s0 { fill: steelblue; }
.bullet .title { font-size: 14px; font-weight: bold; }
.bullet .subtitle { fill: #999; }


</style>
<button>Update</button>
<script type="text/javascript" src="d3/d3.v3.js"></script>
<script src="js/bullet.js"></script>
<script>


var margin = {top: 5, right: 40, bottom: 20, left: 120},
    width = 800 - margin.left - margin.right,
    height = 50 - margin.top - margin.bottom;

var chart = d3.bullet()
    .width(width)
    .height(height);

d3.json("bullet-data.json", function(error, data) {
  var svg = d3.select("body").selectAll("svg")
      .data(data)
    .enter().append("svg")
      .attr("class", "bullet")
      .attr("width", width + margin.left + margin.right)
      .attr("height", height + margin.top + margin.bottom)
    .append("g")
      .attr("transform", "translate(" + margin.left + "," + margin.top + ")")
      .call(chart);

  var title = svg.append("g")
      .style("text-anchor", "end")
      .attr("transform", "translate(-6," + height / 2 + ")");

  title.append("text")
      .attr("class", "title")
      .text(function(d) { return d.title; });
```

```
  title.append("text")
      .attr("class", "subtitle")
      .attr("dy", "1em")
      .text(function(d) { return d.subtitle; });

  d3.selectAll("button").on("click", function() {
    svg.datum(randomize).call(chart.duration(1000));
  });
});

function randomize(d) {
  if (!d.randomizer) d.randomizer = randomizer(d);
  d.markers = d.markers.map(d.randomizer);
  d.measures = d.measures.map(d.randomizer);
  return d;
}

function randomizer(d) {
  var k = d3.max(d.ranges) * .2;
  return function(d) {
    return Math.max(0, d + k * (Math.random() - .5));
  };
}


</script>
</body>
```

This code is a derivative of one of Mike Bostock's blocks here[239]. The full code for this graph can also be found on github[240] or in the code samples bundled with this book (bullet-simple.html, bullet.js and bullet-data.json). A live example can be found on bl.ocks.org[241].

It will become clearer in the process of going through the code below, but as a teaser, it is worth noting that while the code that we will modify is as presented above, we are employing a separate script `bullet.js` to enable the charts.

The first block of our code is the start of the file and sets up our HTML.

[239] http://bl.ocks.org/mbostock/4061961
[240] https://gist.github.com/d3noob/5886992
[241] http://bl.ocks.org/d3noob/5886992

```
<!DOCTYPE html>
<meta charset="utf-8">
<style>
```

This leads into our style declarations.

```
body {
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
  margin: auto;
  padding-top: 40px;
  position: relative;
  width: 800px;
}

button {
  position: absolute;
  right: 40px;
  top: 10px;
}

.bullet { font: 10px sans-serif; }
.bullet .marker { stroke: #000; stroke-width: 2px; }
.bullet .tick line { stroke: #666; stroke-width: .5px; }
.bullet .range.s0 { fill: #eee; }
.bullet .range.s1 { fill: #ddd; }
.bullet .range.s2 { fill: #ccc; }
.bullet .measure.s0 { fill: steelblue; }
.bullet .title { font-size: 14px; font-weight: bold; }
.bullet .subtitle { fill: #999; }
```

We declare the (general) styling for the chart page in the first instance and then the button. Then we move on to the more interesting styling for the bullet charts.

The first line `.bullet { font: 10px sans-serif; }` sets the font size.

The second line sets the colour and width of the symbol marker. So if we were to change it to...

```
.bullet .marker { stroke: red; stroke-width: 10px; }
```

... the result is...

Symbol Marker

The next three lines set the colours for the fill of the qualitative ranges.

```
.bullet .range.s0 { fill: #eee; }
.bullet .range.s1 { fill: #ddd; }
.bullet .range.s2 { fill: #ccc; }
```

You can have more or fewer ranges set here, but to use them you also need the appropriate values in your data file. We will explore how to change this later.

The next line designates the colour for the value being measured.

```
.bullet .measure.s0 { fill: steelblue; }
```

Like the qualitative ranges, we can have more of them, but in my personal opinion, it starts to get a bit confusing.

The final two lines lay out the styling for the label.

The next block of code loads the JavaScript files.

```
</style>
<button>Update</button>
<script src="http://d3js.org/d3.v3.min.js"></script>
<script src="bullet.js"></script>
<script>
```

In this case it's d3 and `bullet.js`. We need to load `bullet.js` as a separate file since it exists outside the code base of the d3.js 'kernel'.

Then we get into the JavaScript. The first thing we do is define the size of the area that we'll be working in.

```
var margin = {top: 5, right: 40, bottom: 20, left: 120},
    width = 800 - margin.left - margin.right,
    height = 50 - margin.top - margin.bottom;
```

Then we define the chart size using the variables that we have just set up.

```
var chart = d3.bullet()
    .width(width)
    .height(height);
```

The other important thing that occurs while setting up the chart is that we use the `d3.bullet` function call to do it. The `d3.bullet` function is the part that resides in the `bullet.js` file that we loaded earlier. The internal workings of `bullet.js` are a window into just how developers are able to craft extra code to allow additional functionality for d3.js.

Then we load our JSON data with our values that we want to display.

```
d3.json("bullet-data.json", function(error, data) {
```

The next block of code is the most important IMHO, since this is where the chart is drawn.

```
var svg = d3.select("body").selectAll("svg")
    .data(data)
  .enter().append("svg")
    .attr("class", "bullet")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
  .append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")")
    .call(chart);
```

However, to look at it you can be forgiven for wondering if it's doing anything at all.

We use our `.select` and `.selectAll` statements to designate where the chart will go (`d3.select("body").selectA` and then load the data as `data` (`.data(data)`).

We add in a svg element (`.enter().append("svg")`) and assign the styling from our css section (`.attr("class", "bullet")`).

Then we set the size of the svg container for an individual bullet chart using `.attr("width", width + margin.left + margin.right)` and `.attr("height", height + margin.top + margin.bottom)`.

We then group all the elements that make up each individual bullet chart with `.append("g")` before placing the group in the right place with `.attr("transform", "translate(" + margin.left + "," + margin.top + ")")`.

Then we wave the magic wand and call the `chart` function with `.call(chart);` which will take all the information from our data file (like the `ranges`, `measures` and `markers` values) and use the `bullet.js` script to create a chart.

The reason I made the comment about the process looking like magic is that the vast majority of the heavy lifting is done by the `bullet.js` file. Because it's abstracted away from the immediate code that we're writing, it looks simplistic, but like all good things, there needs to be a lot of complexity to make a process look simple.

We then add the titles.

```
  var title = svg.append("g")
      .style("text-anchor", "end")
      .attr("transform", "translate(-6," + height / 2 + ")");

  title.append("text")
      .attr("class", "title")
      .text(function(d) { return d.title; });

  title.append("text")
      .attr("class", "subtitle")
      .attr("dy", "1em")
      .text(function(d) { return d.subtitle; });
```

We do this in stages. First we create a variable `title` which will append objects to the grouped element created above (`var title = svg.append("g")`). We apply a style (`.style("text-anchor", "end")`) and transform to the objects (`.attr("transform", "translate(-6," + height / 2 + ")");`).

Then we append the `title` and `subtitle` data (from our JSON file) to our chart with a modicum of styling and placement.

Then we add a button and functions which do the job of applying random data to our variables every time it's pressed.

```
  d3.selectAll("button").on("click", function() {
    svg.datum(randomize).call(chart.duration(1000));
  });
});

function randomize(d) {
  if (!d.randomizer) d.randomizer = randomizer(d);
  d.markers = d.markers.map(d.randomizer);
  d.measures = d.measures.map(d.randomizer);
  return d;
}

function randomizer(d) {
  var k = d3.max(d.ranges) * .2;
  return function(d) {
    return Math.max(0, d + k * (Math.random() - .5));
  };
}
```

I'm not going to delve into the working of the randomize function, because it exists simply to demonstrate the dynamic nature of the chart and not really how the chart is drawn.

However, I will be going through a process later to ensure that we can update the data and the chart automatically which will hopefully be more orientated to practical applications.

That's it! Now we'll go through how you can use the data to change aspects of the chart and what parts of the code need to be adjusted to work with those changes.

Adapting and changing bullet chart components

This section explores some of the simple changes that can be made to bullet charts that may not necessarily be obvious.

Understand your data

The first point to note is that understanding the data loaded from the JSON file is a key to knowing what your chart is going to do.

We'll start by looking at our data in a way that hopefully makes the most sense.

You may be faced with data for a bullet chart that's in a format as follows;

```
[
{"title":"CPU Load","subtitle":"GHz","ranges":[1500,2250,3000],"measures":
[2200],"markers":[2500]},
{"title":"Memory Used","subtitle":"MBytes","ranges":[256,512,1024],"measures":
[768],"markers":[900]}
]
```

This is perfectly valid data, but we'll find it slightly easier to understand if we show it like this…

```
[
  {
    "title":"CPU Load",
    "subtitle":"GHz",
    "ranges":[1500,2250,3000],
    "measures":[2200],
    "markers":[2500]
  },
  {
    "title":"Memory Used",
    "subtitle":"MBytes",
    "ranges":[256,512,1024],
    "measures":[768],
    "markers":[900]
  }
]
```

The data is exactly the same (in terms of content) but I find it a lot easier to comprehend what's going on with the second example.

I have a section in the book called 'Understanding JavaScript Object Notation (JSON)' in the 'Assorted Tips and Tricks' chapter. I found life a lot easier once I started to understand how data was structured in JSON, and if you take a bit of time to understand it, I think you'll find life easier too.

Add as many individual charts as you want.

The example data in the file is an array of two groups. Each group represents the information required to generate one bullet chart. Therefore the example data above will create the following charts;

Two Bullet Charts

You don't need to make any changes to your code in order to add more individual charts. You just need to add more data groups to your JSON file. The following example uses exactly the same code, but with several extra groups of data.

Lots of Bullet Charts

Add more ranges and measures

Returning to our single chart example, you can see from the JSON data that there are three specified ranges and one measure.

```
[
  {
    "title":"CPU 1 Load",
    "subtitle":"GHz",
    "ranges":[1500,2250,3000],
    "measures":[2200],
    "markers":[2500]
  }
]
```

The same was true for the css in the JavaScript code. Three ranges and one measure

```
.bullet { font: 10px sans-serif; }
.bullet .marker { stroke: #000; stroke-width: 2px; }
.bullet .tick line { stroke: #666; stroke-width: .5px; }
.bullet .range.s0 { fill: #eee; }
.bullet .range.s1 { fill: #ddd; }
.bullet .range.s2 { fill: #ccc; }
.bullet .measure.s0 { fill: steelblue; }
.bullet .title { font-size: 14px; font-weight: bold; }
.bullet .subtitle { fill: #999; }
```

By matching the css for the `.bullet` style with the data you can add more or fewer of both. For example here's example data, css and a chart with five ranges and two measures.

```
[
  {
    "title":"CPU 1 Load",
    "subtitle":"GHz",
    "ranges":[500,1000,1500,2250,3000],
    "measures":[1250, 2200],
    "markers":[2650]
  }
]
```

```
.bullet { font: 10px sans-serif; }
.bullet .marker { stroke: lightgreen; stroke-width: 5px; }
.bullet .tick line { stroke: #666; stroke-width: .5px; }
.bullet .range.s0 { fill: navy; }
.bullet .range.s1 { fill: mediumblue; }
.bullet .range.s2 { fill: dodgerblue; }
.bullet .range.s3 { fill: aqua; }
.bullet .range.s4 { fill: lightblue; }
.bullet .measure.s0 { fill: red; }
.bullet .measure.s1 { fill: pink; }
.bullet .title { font-size: 14px; font-weight: bold; }
.bullet .subtitle { fill: #999; }
```

Bullet Chart with Five Ranges and Two Measures

First of all. Yes, I know the colours are gaudy. Hopefully they stand out. Don't abuse your own graphs in this hideous way.

More importantly though, you can now get a better idea of how to align the range and measure values in the JSON file with the `.range` and `.measure` styles in the css.

Bullet Chart with Five Ranges and Two Measures

The diagram shows that the `.range` and `.measure` bars are numbered from the right. (for example the 'navy' colour showing the range up to 3000 GHz is designated `.range.s0`. At first this convention of numbering from the right confused me. I imagined that the smallest range should be `.range.s0` and this should be on the left. Then I realised that the numbering related to the *layer* of the range. So this would make `.range.s0` go from 0 to 3000. Then the second layer would be `.range.s1` which would go *on top of* `.range.s0` from 0 to 2250, thereby covering most of `.range.s0` except for the part that exceeded `.range.s1`. Which is exactly what we see with successively higher layers having higher numbers. The same is true for the `.measure` numbers and layers.

Updating a bullet chart automatically

Displaying static data is a good start for a bullet chart, but if you have data that's changing dynamically, you need to be able to re-load the information and display it automatically.

To adapt our code to this purpose we will first remove the parts that added the button.

Remove this portion from the css section;

```
button {
  position: absolute;
  right: 40px;
  top: 10px;
}
```

Then remove this line that added the button in the html section;

```
<button>Update</button>
```

All we need to do now is change the section that called the original json file from;

```
d3.json("bullet-data.json", function(error, data) {
```

... to ...

```
d3.json("bullet-data2.json", function(error, data) {
```

So that we're dealing with a different json file (there's no need to go messing around with our original data).

Change the section that used to call the function to randomise the data with the button click from...

```
  d3.selectAll("button").on("click", function() {
    svg.datum(randomize).call(chart.duration(1000));
  });
```

... to ...

```
  setInterval(function() {
                updateData();
        }, 1000);
```

This new piece of code simply sets up a repeating function that calls another function (`updateData`) every 1000ms.

The final change is to replace the original functions that randomised the data...

```
function randomize(d) {
  if (!d.randomizer) d.randomizer = randomizer(d);
  d.markers = d.markers.map(d.randomizer);
  d.measures = d.measures.map(d.randomizer);
  return d;
}

function randomizer(d) {
  var k = d3.max(d.ranges) * .2;
  return function(d) {
    return Math.max(0, d + k * (Math.random() - .5));
  };
}
```

... with our new function that updates the data ...

```
function updateData() {
        d3.json("bullet-data2.json", function(error, data) {
                d3.select("body").selectAll("svg")
                        .datum(function (d, i) {
                                d.ranges = data[i].ranges;
                                d.measures = data[i].measures;
                                d.markers = data[i].markers;
                                return d;
                        })
                        .call(chart.duration(1000));
        });
}
```

This new function (`updateData`) reads in our json file again, selects all the svg elements then updates all the `.ranges`, `.measures` and `.markers` data with whatever was in the file. Then it calls the `chart` function that updates the bullet charts.

All the code components for this script can be found on github[242] or in the code samples bundled with this book (bullet-auto.html and bullet-data2.json). A live example can be found on bl.ocks.org[243] (although it won't update since the data file can't be updated online).

[242] https://gist.github.com/d3noob/5893649
[243] http://bl.ocks.org/d3noob/5893649

Mapping with d3.js

Another string to the bow of d3.js is the addition of a set of powerful routines for handling geographical information.

In the same sense that a line graph is a simple representation of data on a document, a map can be regarded as a set of points with an underlying coordinate system. When you say it like that it seems obvious that it should be applied as a document for display. However, I don't want to give the impression that this is some sort of trivial matter for either the original developers or for you, the person who wants to display a map. Behind the scenes for this type of work, the thought that must have gone into making the code usable and extensible must have been enormous.

Mike Bostock has lauded the work of Jason Davies in the development of the latest major version of d3.js (version 3) for his work on improving mapping capability. A visit to his home page[244] provides a glimpse into Jason's expertise and no visit would be complete without marvelling at his work with geographic projections[245].

Examples

I am firmly of the belief that mapping in particular has an enormous potential for adding value to data sets. The following collection of examples gives a brief taste of what has been accomplished by combining geographic information and D3 thus far. (The screen shots following have been sourced from the biovisualize gallery[246] and as such provide attribution to the best of my ability. If I have incorrectly attributed the source or author please let me know and I will correct it promptly.)

Faux D3 3d globe integrated with Mapbox / Open Street Map

Above is an interactive visualization showing the position of the main map on a faux D3 3d globe with a Mapbox / Open Street Map main window. Source dev.geosprocket.com[247] Source Bill Morris.

[244] http://www.jasondavies.com/
[245] http://www.jasondavies.com/maps/
[246] http://biovisualize.github.com/d3visualization/#visualizationType=map
[247] http://dev.geosprocket.com/d3/finder/

Kentucky Count Population from the 2010 census

This is a breakdown of population in Kentucky Counties from the 2010 census. Source: ccarpenterg.github.com[248] by Cristian Carpenter.

Beijing air pollution

This map visualizes air pollution in Beijing. Source: scottcheng.github.com[249] by Scott Cheng.

[248] http://ccarpenterg.github.com/blog/us-census-visualization-with-d3js/
[249] http://scottcheng.github.com/bj-air-vis/

Shuttle Radar Topography Mission tile downloading

This is a section of the globe that is presented on the Shuttle Radar Topography Mission tile downloading web site. This excellent site uses the interactive globe to make the selection of SRTM tiles easy. Source dwtkns.com[250] by Derek Watkins.

Animated World tour

This is a static screen-shot of an animated tour of the Worlds countries. Source bl.ocks.org[251] by Mike Bostock.

[250] http://dwtkns.com/srtm/
[251] http://bl.ocks.org/mbostock/4183330

A Chicago Divided by Killings: New Your Times

This is one of the great infographics published by the New York Times[252]. Source: www.nytimes.com[253] by Mike Bostock, Shan Carter and Kevin Quealy.

Concentric circles emanating from glowing red dot

This is an animated graphic showing a series of concentric circles emanating from glowing red dot which was styled after a news article in The Onion[254]. Source: bl.ocks.org[255] by Mike Bostock.

[252] http://www.nytimes.com
[253] http://www.nytimes.com/interactive/2013/01/02/us/chicago-killings.html?_r=0
[254] http://www.theonion.com/video/breaking-news-series-of-concentric-circles-emanati,14204/
[255] http://bl.ocks.org/mbostock/4503672

Christchurch earthquakes timeline

Here we see earthquakes represented on a selectable timeline where D3 generates a svg overlay and the map layer is created using Leaflet. Source: bl.ocks.org[256] by tnightingale.

Earthquakes in the past 24 hours

Carrying on with the earthquake theme, this is a map of all earthquakes in the past 24 hours over magnitude 2.5. Source: bl.ocks.org[257] by benelsen.

[256] http://bl.ocks.org/tnightingale/4718717
[257] http://bl.ocks.org/benelsen/4969007

Satellite projection

An interactive satellite projection. Source dev.geosprocket.com[258] by Bill Morris.

GeoJSON and TopoJSON

Projecting countries and various geographic features onto a map can be a very data hungry exercise. By that I mean that the information required to present geographic shapes can result in data files that are quite large. GeoJSON has been the default geographic data file of choice for quite some time, and as the name would suggest it encodes the data in a JSON type hierarchy. Often these GeoJSON files include a significant amount of extraneous detail or incorporate a level of accuracy that is impractical (too detailed).

Enter TopoJSON. Mike Bostock has designed TopoJSON as an extension to GeoJSON in the sense that it has a similar structure, but the geometries are not encoded discretely and where they share features, they are combined. Additionally TopoJSON encodes numeric values more efficiently and can incorporate a degree of simplification. This simplification can result in savings of file size of 80% or more depending on the area and use of compression. Although TopoJSON has only begun to be used, the advantages of it seem clear and so I will anticipate its future use by incorporating it in my example diagrams (not that the use of GeoJSON differs much if at all). A great description of TopoJSOn can be found on the TopoJSON wiki on github[259].

[258] http://dev.geosprocket.com/d3/sat/
[259] https://github.com/mbostock/topojson/wiki

Starting with a simple map

Our starting example will demonstrate the simple display of a World map. Our final result will looks like this;

The World

The data file for the World map is one produced by Mike Bostock's as part of his TopoJSON work.

We'll move through the explanation of the code in a similar process to the one we went through when highlighting the function of the Sankey diagram. Where there are areas that we have covered before, I will gloss over some details on the understanding that you will have already seen them explained in an earlier section (most likely the basic line graph example).

The full code for this graphic can be found on github[260] or in the code samples bundled with this book (world-map.html and world-110m2). A live example can be found on bl.ocks.org[261].

Here is the full code;

```
<!DOCTYPE html>
<meta charset="utf-8">
<style>
```

[260] https://gist.github.com/d3noob/5189184
[261] http://bl.ocks.org/d3noob/5189184

```
path {
  stroke: white;
  stroke-width: 0.25px;
  fill: grey;
}


</style>
<body>
<script src="http://d3js.org/d3.v3.min.js"></script>
<script src="http://d3js.org/topojson.v0.min.js"></script>
<script>


var width = 960,
    height = 500;

var projection = d3.geo.mercator()
    .center([0, 5 ])
    .scale(150)
    .rotate([-180,0]);

var svg = d3.select("body").append("svg")
    .attr("width", width)
    .attr("height", height);

var path = d3.geo.path()
    .projection(projection);

var g = svg.append("g");

// load and display the World
d3.json("world-110m2.json", function(error, topology) {
    g.selectAll("path")
      .data(topojson.object(topology, topology.objects.countries)
          .geometries)
    .enter()
      .append("path")
      .attr("d", path)
});


</script>
</body>
</html>
```

One of the first things that struck me when I first saw the code to draw a map was how small it was (the amount of code, not the World). It's a measure of the degree of abstraction that D3 is able to provide to the process of getting data from a raw format to the screen that such a complicated task can be condensed to such an apparently small amount of code. Of course that doesn't tell the whole story. Like a duck on a lake, above the water all is serene and calm while below the water the feet are paddling like fury. In this case, our code looks serene because D3 is doing all the hard work :-).

The first block of our code is the start of the file and sets up our HTML.

```
<!DOCTYPE html>
<meta charset="utf-8">
<style>
```

This leads into our style declarations.

```
path {
  stroke: white;
  stroke-width: 0.25px;
  fill: grey;
}
```

We only state the properties of the path components which will make up our countries. Obviously we will fill them with grey and have a thin (`0.25px`) line around each one.

The next block of code loads the JavaScript files.

```
</style>
<body>
<script src="http://d3js.org/d3.v3.min.js"></script>
<script src="http://d3js.org/topojson.v0.min.js"></script>
<script>
```

In this case it's d3 and topojson. We load `topojson.v0.min.js` as a separate file because it's still fairly new. In other words it hasn't been incorporated into the main d3.js code base (that's an assumption on my part since it might exist in isolation or perhaps end up as a plug-in). Whatever the case, for the time being, it exists as a separate file.

Then we get into the JavaScript. The first thing we do is define the size of our map.

```
var width = 960,
    height = 500;
```

Then we get into one of the simple, but cool parts of making any map. Setting up the view.

```
var projection = d3.geo.mercator()
    .center([0, 5 ])
    .scale(150)
    .rotate([-180,0]);
```

The projection is the way that the geographic coordinate system is adjusted for display on our flat screen. The screen is after all a two dimensional space and we are trying to present a three dimensional object. This is a big deal to cartographers in the sense that selecting a geographic projection for a map is an exercise in compromise. You can make it look pretty, but in doing so you can grievously distort the land size / shape. On the other hand you might make it more accurate, in size / shape but people will have trouble recognising it because they're so used to the standard Mercator projection. For example, the awesome Waterman Butterfly[262].

The Waterman Butterfly

There are a lot of alternatives available. Please have a browse on the wiki[263] where you will find a huge range of options (66 at time of writing).

In our case we've gone with the conservative Mercator option.

Then we define three aspects of the projection. Center, scale and rotate.

[262] http://bl.ocks.org/mbostock/4458497
[263] https://github.com/mbostock/d3/wiki/Geo-Projections

center

If center is specified, this sets the projection's center to the specified location as a two-element array of longitude and latitude in degrees and returns the projection. If center is not specified the default of (0°,0°) is used.

Our example is using `[0, 5 ]` which I have selected as being in the middle (I use `0`) for longitude (left to right) and `5` degrees North of the equator (for latitude, North is positive, South is negative). This was purely to make it look aesthetically pleasing. Here's the result of setting the center to `[100,30]`.

Center set to `[100,30]`

The map has been centered on 100 degrees West and 30 degrees North. Of course, it's also been pushed to the left without the right hand side of the map scrolling around. We'll get to that in a moment.

scale

If scale is specified, this sets the projection's scale factor to the specified value. If scale is not specified, it returns the current scale factor which defaults to 150. It's important to note that scale factors are not consistent across projections.

Our current map uses a scale of 900. Again, this has been set for aesthetics. Keeping our center of `[100,30]`, if we increase our scale to `2000` this is the result.

Scale set to `2000`

rotate

If rotation is specified, this sets the projection's three-axis rotation to the specified angles for yaw, pitch and roll (equivalently longitude, latitude and roll) in degrees and returns the projection. If rotation is not specified, it sets the values to [0, 0, 0]. If the specified rotation has only two values, rather than three, the roll is assumed to be 0°.

In our map we have specified `[-180,0]` so we can assume a roll value of zero. Likewise we have rotated our map by -180 degrees in longitude. This has been done specifically to place the map with the center on the anti-meridian (The international date line in the middle of the Pacific Ocean). If we return the value to `[0,0]`(with our original values of `scale` and `center` this is the result.

Rotate set to `[0,0]`

In this case the centre of the map lines up with the meridian.

The next block of code sets our svg window;

```
var svg = d3.select("body").append("svg")
    .attr("width", width)
    .attr("height", height);
```

The following portion of code creates a new geographic path generator;

```
var path = d3.geo.path()
    .projection(projection);
```

The path generator (`d3.geo.path()`) is used to specify a projection type (`.projection`) which was defined earlier as a Mercator projection via the variable `projection`. (I'm not entirely sure,

but it is possible that I have just set some kind of record for use of the word 'projection' in a sentence.)

We then declare `g` as our appended svg.

```
var g = svg.append("g");
```

The last block of JavaScript draws our map.

```
d3.json("world-110m2.json", function(error, topology) {
    g.selectAll("path")
      .data(topojson.object(topology, topology.objects.countries)
          .geometries)
    .enter()
      .append("path")
      .attr("d", path)
});
```

We load the TopoJSON file with the coordinates for our World map (`world-110m2.json`). Then we declare that we are going to act on all the `path` elements in the graphic (`g.selectAll("path")`).

Then we pull the data that defines the countries from the TopoJSON file (`.data(topojson.object(topology, topology.objects.countries).geometries)`). We add it to the data that we're going to display (`.enter()`) and then we append that data as `path` elements (`.append("path")`).

The last html block closes off our tags and we have a map!

The World map centered on the Pacific

Zooming and panning a map

With our map displayed nicely we need to be able to move it about to explore it fully . To do this we can provide the functionality to zoom and pan it using the mouse.

Towards the end of the script, just before the close off of the script at the `</script>` tag we can add in the following code;

```
var zoom = d3.behavior.zoom()
    .on("zoom",function() {
        g.attr("transform","translate("+
            d3.event.translate.join(",")+")scale("+d3.event.scale+")");
        g.selectAll("path")
            .attr("d", path.projection(projection));
});

svg.call(zoom)
```

This block of code introduces the `behaviors` functions. Using the `d3.behavior.zoom` function creates event listeners (which are like hidden functions standing by to look out for a specific type of activity on the computer and in this case mouse actions) to handle zooming and panning gestures on a container element (in this case our map). More information on the range of zoom options is available on the D3 Wiki[264].

We begin by declaring the `zoom` function as `d3.behavior.zoom`.

Then we instruct the computer that when it 'sees' a 'zoom' event to carry out another function (`.on("zoom",function() {`).

That function firstly gathers the (correctly formatted) `translate` and `scale` attributes in...

```
        g.attr("transform","translate("+
            d3.event.translate.join(",")+")scale("+d3.event.scale+")");
```

... and then applies them to all the path elements (which are the shapes of the countries) via...

```
        g.selectAll("path")
            .attr("d", path.projection(projection));
```

Lastly we call the zoom function.

```
svg.call(zoom)
```

Then we relax and explore our map!

[264] https://github.com/mbostock/d3/wiki/Zoom-Behavior

The World map with zoom and pan

The full code for this graphic can be found on github[265] or in the code samples bundled with this book (world-map-zoom-pan.html and world-110m2). A live example can be found on bl.ocks.org[266].

Displaying points on a map

Displaying maps and exploring them is pretty entertaining, but as anyone who has participated in the improvement of our geographic understanding of our world via projects such as Open Street Map[267] will tell you, there's a whole new level of cool to be attained by adding to a map.

With that in mind, our next task is to add some simple detail in the form of points that show the location of cities.

To do this we will load in a csv file with data that identifies our cities and includes latitude and longitude details. Our file is called `cities.csv` and looks like this;

```
code,city,country,lat,lon
ZNZ,ZANZIBAR,TANZANIA,-6.13,39.31
TYO,TOKYO,JAPAN,35.68,139.76
AKL,AUCKLAND,NEW ZEALAND,-36.85,174.78
BKK,BANGKOK,THAILAND,13.75,100.48
DEL,DELHI,INDIA,29.01,77.38
SIN,SINGAPORE,SINGAPOR,1.36,103.75
BSB,BRASILIA,BRAZIL,-15.67,-47.43
RIO,RIO DE JANEIRO,BRAZIL,-22.90,-43.24
```

[265] https://gist.github.com/d3noob/5189284
[266] http://bl.ocks.org/d3noob/5189284
[267] http://www.openstreetmap.org/

```
YTO,TORONTO,CANADA,43.64,-79.40
IPC,EASTER ISLAND,CHILE,-27.11,-109.36
SEA,SEATTLE,USA,47.61,-122.33
```

While we're only going to use the latitude and longitude for our current work, the additional details could just as easily be used for labelling or tooltips.

We need to place our code carefully in this case because while you might have some flexibility in getting the right result with a locally hosted version of a map, there is a possibility that with a version hosted in the outside World (*gasp* the internet) you could strike trouble.

The code to load the cities should be placed inside the function that is loading the World map as indicated below;

```
d3.json("world-110m2.json", function(error, topology) {
    g.selectAll("path")
      .data(topojson.object(topology, topology.objects.countries)
          .geometries)
    .enter()
      .append("path")
      .attr("d", path)
                            // <== Put the new code block here
});
```

Here's the new code;

```
    d3.csv("cities.csv", function(error, data) {
        g.selectAll("circle")
           .data(data)
           .enter()
           .append("circle")
           .attr("cx", function(d) {
                  return projection([d.lon, d.lat])[0];
           })
           .attr("cy", function(d) {
                  return projection([d.lon, d.lat])[1];
           })
           .attr("r", 5)
           .style("fill", "red");
```

We'll go through the code and then explain the quirky thing about it.

First of all we load the `cities.csv` file (`d3.csv("cities.csv", function(error, data) {`). Then we select all the circle elements (`g.selectAll("circle")`), assign our data (`.data(data)`), enter our data (`.enter()`) and then add in circles (`.append("circle")`).

Then we set the x and y position for the circles based on the longitude (`([d.lon, d.lat])[0]`) and latitude (`([d.lon, d.lat])[1]`) information in the csv file.

Finally we assign a radius of 5 pixels and fill the circles with red.

The quirky thing about the new code block is that we have to put it inside the code block that loads the world data (`d3.json("world-110m2.json", function(error, topology) {`). We could place the two blocks one after the others (load / draw the world data, then load / draw the circles). And this will probably work if you run the file from your local computer. But when you host the files on the internet, it takes too long to load the world data compared to the city data and the end result is that the city data gets drawn before the world data and this is the result.

The cities under the World

To avoid the problem we place the loading of the city data *into* the code that loads the World data. That way the city data doesn't get loaded until the World data is loaded and then the circles get drawn on top of the world instead of under it :-).

The cities on top of the World

The full code for this graphic can be found on github[268] or in the code samples bundled with this book (world-map-cities.html, cities.csv and world-110m2). A live example can be found on bl.ocks.org[269].

Additionally the full code can be found in the appendix section at the rear of the book.

As an added extra and in response to a question that was asked on the d3noob.org blog, the names of the cities can be placed alongside the location dots by the addition of the following block of code inside the 'cities' loading portion of the script;

```
g.selectAll("text")
   .data(data)
   .enter()
 .append("text") // append text
   .attr("x", function(d) {
           return projection([d.lon, d.lat])[0];
   })
   .attr("y", function(d) {
           return projection([d.lon, d.lat])[1];
   })
   .attr("dy", -7) // set y position of bottom of text
  .style("fill", "black") // fill the text with the colour black
  .attr("text-anchor", "middle") // set anchor y justification
  .text(function(d) {return d.city;}); // define the text to display
```

The end result shows the name of the cities placed above and centred with respect to the location.

[268] https://gist.github.com/d3noob/5193723
[269] http://bl.ocks.org/d3noob/5193723

The cities on top of the World

The full code for this graphic can be found on github[270] or in the code samples bundled with this book (world-map-cities-text.html, cities.csv and world-110m2). A live example can be found on bl.ocks.org[271].

[270] https://gist.github.com/d3noob/401237468c9e38cea8c7
[271] http://bl.ocks.org/d3noob/401237468c9e38cea8c7

Making maps with d3.js and leaflet.js combined

If you've read to this point in D3 Tips and Tricks, you may be aware that I have also written another book called 'Leaflet Tips and Tricks[272]'. I haven't written both books because they are integrated with each other or because they seem made to compliment each other. I wrote them because both libraries are the best of breed (IMHO) at what they do. It should come as little surprise that they can have a lot to offer users who want to combine the incredible scope of d3.js's data manipulation functions and the elegance of leaflet.js's tile map presentation capabilities.

leaflet.js Overview

Leaflet.js[273] is an Open Source JavaScript library designed to make deploying maps on a web page easy. It uses a paltry 34kB (at time of writing) JavaScript file that loads with your web page and provides access to a range of functions that will allow you to present a map.

Its goals are to be simple to use while focussing on performance and usability, but it's also built to be extended using plugins that extend its functionality. It has an excellent API which is well documented, so there are no mysteries to using it successfully in a range of situations.

Out of the box Leaflet provides the functionality to add markers, popups, overlay lines and shapes, use multiple layers, zoom, pan and generally have a good time :-). But these are just the the *core* features of Leaflet. One of the significant strengths of Leaflet is the ability to extend the functionality of the script with plugins from third parties. At the time of writing there are over 80 separate plugins that allow features such as overlaying a heatmap, animating markers, loading csv files of data, drawing complex shapes, measuring distance, manipulating layers and displaying coordinates.

Leaflet is simple, elegant and functional but powerful. There's a good chance that even if you don't present maps with Leaflet, you'll be using ones that someone else made with it at some stage on the Internet.

Why use leaflet.js when d3.js does maps too?

Good question. I can see you've been paying attention.

There is a significant difference between the underlying way that d3.js and leaflet.js presents mapping data. D3.js predominantly focusses on vector based graphics when drawing maps and leaflet.js leverages the huge range of bitmap based map tiles that are available for use around the world. Both bitmap and vector based solutions have strengths and weaknesses depending on the application. Combining both allows the use of the best of both worlds.

[272] https://leanpub.com/leaflet-tips-and-tricks
[273] http://leafletjs.com/

Leaflet map with d3.js objects that scale with the map

The first example we'll look at will project a leaflet.js map on the screen with a d3.js object (in this case a simple rectangle) onto the map.

The rectangle will be bound to a set of geographic coordinates so that as the map is panned and zoomed the rectangle will shrink and grow. For example the following diagram shows a rectangle (made with d3.js) superimposed over a leaflet.js map;

Rectangular d3 area on leaflet map

If we then zoom in…

Zoomed rectangular d3 area on leaflet map

...the rectangle zooms in as well.

This may not sound terribly exciting and if you're familiar with Leaflet you will know that it is possible to draw polygons onto a map using only leaflet's built in functions. However, the real strength of this application of vector data comes when making the d3.js content interactive which is more difficult with leaflet.js.

For an excellent example of this please visit Mike Bostock's tutorial[274] where he demonstrates superimposing a map of the United States separated by state (which react individually to the mouse being hovered over them). My following explanation is a humble derivation of his code.

Speaking of code, here is a full listing of the code that we will be using;

```
<!DOCTYPE html>
<html>
<head>
    <title>Leaflet and D3 Map</title>
    <meta charset="utf-8" />
    <link
        rel="stylesheet"
        href="http://cdn.leafletjs.com/leaflet-0.7/leaflet.css"
    />

</head>
<body>

    <div id="map" style="width: 600px; height: 400px"></div>
```

[274] http://bost.ocks.org/mike/leaflet/

```
    <script src="http://d3js.org/d3.v3.min.js"></script>

<script
    src="http://cdn.leafletjs.com/leaflet-0.7/leaflet.js">
</script>

<script>

    var map = L.map('map').setView([-41.2858, 174.7868], 13);
    mapLink =
        '<a href="http://openstreetmap.org">OpenStreetMap</a>';
    L.tileLayer(
        'http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
        attribution: '&copy; ' + mapLink + ' Contributors',
        maxZoom: 18,
        }).addTo(map);

            // Add an SVG element to Leaflet's overlay pane
            var svg = d3.select(map.getPanes().overlayPane).append("svg"),
                    g = svg.append("g").attr("class", "leaflet-zoom-hide");

            d3.json("rectangle.json", function(geoShape) {

            //  create a d3.geo.path to convert GeoJSON to SVG
            var transform = d3.geo.transform({point: projectPoint}),
        path = d3.geo.path().projection(transform);

            // create path elements for each of the features
            d3_features = g.selectAll("path")
                    .data(geoShape.features)
                    .enter().append("path");

            map.on("viewreset", reset);

            reset();

            // fit the SVG element to leaflet's map layer
            function reset() {

                    bounds = path.bounds(geoShape);

                    var topLeft = bounds[0],
                            bottomRight = bounds[1];

                    svg .attr("width", bottomRight[0] - topLeft[0])
                            .attr("height", bottomRight[1] - topLeft[1])
```

```
                                .style("left", topLeft[0] + "px")
                                .style("top", topLeft[1] + "px");

                        g .attr("transform", "translate(" + -topLeft[0] + ","
                                                          + -topLeft[1] + ")");

                        // initialize the path data
                        d3_features.attr("d", path)
                                .style("fill-opacity", 0.7)
                                .attr('fill','blue');
                }

                // Use Leaflet to implement a D3 geometric transformation.
                function projectPoint(x, y) {
                        var point = map.latLngToLayerPoint(new L.LatLng(y, x));
                        this.stream.point(point.x, point.y);
                }

        })

    </script>
</body>
</html>
```

There is also an associated json data file (called `rectangle.json`) that has the following contents;

```
{
"type": "FeatureCollection",
"features": [ {
        "type": "Feature",
        "geometry": {
                "type": "Polygon",
                "coordinates": [ [
                [ 174.78, -41.29 ],
                [ 174.79, -41.29 ],
                [ 174.79, -41.28 ],
                [ 174.78, -41.28 ],
                [ 174.78, -41.29 ]
                ] ]
                }
        }
]
}
```

The full code and a live example are available online at bl.ocks.org[275] or GitHub[276]. They are

[275] http://bl.ocks.org/d3noob/9211665
[276] https://gist.github.com/d3noob/9211665

also available as the files 'leaflet-d3-combined.html' and 'rectangle.json' as a separate download with D3 Tips and Tricks. A a copy of all the files that appear in the book can be downloaded (in a zip file) when you download the book from Leanpub[277]

While I will explain the code below, please be aware that I will gloss over some of the simpler sections that are covered in other sections of either books and will instead focus on the portions that are important to understand the combination of d3 and leaflet.

Our code begins by setting up the html document in a fairly standard way.

```
<!DOCTYPE html>
<html>
<head>
    <title>Leaflet and D3 Map</title>
    <meta charset="utf-8" />
    <link
        rel="stylesheet"
        href="http://cdn.leafletjs.com/leaflet-0.7/leaflet.css"
    />

</head>
<body>

    <div id="map" style="width: 600px; height: 400px"></div>

        <script src="http://d3js.org/d3.v3.min.js"></script>

    <script
        src="http://cdn.leafletjs.com/leaflet-0.7/leaflet.js">
    </script>
```

Here we're getting some css styling and loading our leaflet.js / d3.js libraries. The only configuration item is where we set up the size of the map (in the `<style>` section and as part of the `map` div).

Then we break into the JavaScript code. The first thing we do is to project our Leaflet map;

```
var map = L.map('map').setView([-41.2858, 174.7868], 13);
mapLink =
        '<a href="http://openstreetmap.org">OpenStreetMap</a>';
L.tileLayer(
        'http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
        attribution: '&copy; ' + mapLink + ' Contributors',
        maxZoom: 18,
        }).addTo(map);
```

[277] https://leanpub.com/leaflet-tips-and-tricks

This is exactly the same as we have done in any of the simple map explanations in Leaflet Tips and Tricks[278] and in this case we are using the OpenStreetMap tiles.

Then we start on the d3.js part of the code.

The first part of that involves making sure that Leaflet and D3 are synchronised in the view that they're projecting. This synchronisation needs to occur in zooming and panning so we add an SVG element to Leaflet's `overlayPlane`

```
var svg = d3.select(map.getPanes().overlayPane).append("svg"),
        g = svg.append("g").attr("class", "leaflet-zoom-hide");
```

Then we add a `g` element that ensures that the SVG element and the Leaflet layer have the same common point of reference. Otherwise when they zoomed and panned it could be offset. The `leaflet-zoom-hide` affects the presentation of the map when zooming. Without it the underlying map zooms to a new size, but the d3.js elements remain as they are until the zoom effect has taken place and then they adjust. It still works fine, but it 'looks' wrong.

Then we load our data file with the line...

```
d3.json("rectangle.json", function(geoShape) {
```

This is pretty standard fare for d3.js but it's worth being mindful that while the type of data file is `.json` this is a GeoJSON file and they have particular features (literally) that allow them to do their magic. There is a good explanation of how they are structured at geojson.org[279] for those who are unfamiliar with the differences.

Using our data we need to ensure that it is correctly transformed from our latitude/longitude coordinates as supplied to coordinates on the screen. We do this by implementing d3's geographic transformation features (d3.geo[280]).

```
var transform = d3.geo.transform({point: projectPoint}),
        path = d3.geo.path().projection(transform);
```

Here the path that we want to create in SVG is generated from the points that are supplied from the data file which are converted by the function `projectPoint` This function (which is placed at the end of the file) takes our latitude and longitudes and transforms them to screen (layer) coordinates[281].

```
function projectPoint(x, y) {
        var point = map.latLngToLayerPoint(new L.LatLng(y, x));
        this.stream.point(point.x, point.y);
}
```

With the transformations now all taken care of we can generate our path in the traditional d3.js way and append it to our `g` group.

[278] https://leanpub.com/leaflet-tips-and-tricks/
[279] http://geojson.org/geojson-spec.html
[280] https://github.com/mbostock/d3/wiki/API-Reference#wiki-d3geo-geography
[281] http://leafletjs.com/reference.html#map-conversion-methods

```
d3_features = g.selectAll("path")
        .data(geoShape.features)
        .enter().append("path");
```

The last 'main' part of our JavaScript makes sure that when our view of what we're looking at changes (we zoom or pan) that our d3 elements change as well;

```
        map.on("viewreset", reset);

        reset();
```

Obviously when our view changes we call the function `reset`. It's the job of the `reset` function to ensure that whatever the leaflet layer does, the SVG (d3.js) layer follows;

```
function reset() {

    bounds = path.bounds(geoShape);

    var topLeft = bounds[0],
        bottomRight = bounds[1];

    svg .attr("width", bottomRight[0] - topLeft[0])
        .attr("height", bottomRight[1] - topLeft[1])
        .style("left", topLeft[0] + "px")
        .style("top", topLeft[1] + "px");

    g .attr("transform", "translate(" + -topLeft[0] + ","
                                      + -topLeft[1] + ")");

    // initialize the path data
    d3_features.attr("d", path)
        .style("fill-opacity", 0.7)
        .attr('fill','blue');
}
```

It does this by establishing the `topLeft` and `bottomRight`corners of the desired area and then it applies the `width`, `height`, `top` and `bottom` attributes to the svg element and translates the `g` element to the right spot. Last, but not least it redraws the path.

The end result being a fine combination of leaflet.js map and ds.js element;

Rectangular d3 area on leaflet map

Leaflet map with d3.js elements that are overlaid on a map

The next example of a combination of d3.js and leaflet.js is one where we want to have an element overlaid on our map at a specific location, but have it remain a specific size over the map. For example, here we will display 5 circles which are centred at specific geographic locations.

d3.js circles fixed in geographic location on leaflet map but constant size

When we zoom out of the map, those circles remain over the geographic location, but the same size on the screen.

Zoomed d3.js circles fixed in geographic location on leaflet map but constant size

You may (justifiably) ask yourself why we would want to do this with d3.js when Leaflet could do the same job with a marker? The answer is that as cool as leaflet.js's markers are, d3 elements have a wider range of features that make their use advantageous in some situations. For instance if you want to animate or rotate the icons or dynamically adjust some of their attributes, d3.js would have a greater scope for adjustments.

The following code draws circles at geographic locations;

```
<!DOCTYPE html>
<html>
<head>
    <title>d3.js with leaflet.js</title>

    <link
        rel="stylesheet"
        href="http://cdn.leafletjs.com/leaflet-0.7/leaflet.css"
    />
    <script src="http://d3js.org/d3.v3.min.js"></script>

    <script
        src="http://cdn.leafletjs.com/leaflet-0.7/leaflet.js">
    </script>

</head>
<body>

    <div id="map" style="width: 600px; height: 400px"></div>
```

```
<script type="text/javascript">

    var map = L.map('map').setView([-41.2858, 174.7868], 13);
    mapLink =
        '<a href="http://openstreetmap.org">OpenStreetMap</a>';
    L.tileLayer(
        'http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
        attribution: '&copy; ' + mapLink + ' Contributors',
        maxZoom: 18,
        }).addTo(map);

// Initialize the SVG layer
map._initPathRoot()

// We pick up the SVG from the map object
var svg = d3.select("#map").select("svg"),
g = svg.append("g");

d3.json("circles.json", function(collection) {
    // Add a LatLng object to each item in the dataset
    collection.objects.forEach(function(d) {
        d.LatLng = new L.LatLng(d.circle.coordinates[0],
                                d.circle.coordinates[1])
    })

    var feature = g.selectAll("circle")
        .data(collection.objects)
        .enter().append("circle")
        .style("stroke", "black")
        .style("opacity", .6)
        .style("fill", "red")
        .attr("r", 20);

    map.on("viewreset", update);
    update();

    function update() {
        feature.attr("transform",
        function(d) {
            return "translate("+
                map.latLngToLayerPoint(d.LatLng).x +","+
                map.latLngToLayerPoint(d.LatLng).y +")";
            }
        )
    }
})
```

```
</script>
</body>
</html>
```

There is also an associated json data file (called `circles.json`) that has the following contents;

```
{"objects":[
{"circle":{"coordinates":[-41.28,174.77]}},
{"circle":{"coordinates":[-41.29,174.76]}},
{"circle":{"coordinates":[-41.30,174.79]}},
{"circle":{"coordinates":[-41.27,174.80]}},
{"circle":{"coordinates":[-41.29,174.78]}}
]}
```

The full code and a live example are available online at bl.ocks.org[282] or GitHub[283]. They are also available as the files 'leaflet-d3-linked.html' and 'circles.json' as a separate download with D3 Tips and Tricks. A a copy of all the files that appear in the book can be downloaded (in a zip file) when you download the book from Leanpub[284]

While I will explain the code below, as with the previous example (which is similar, but different) please be aware that I will gloss over some of the simpler sections that are covered in other sections of either books and will instead focus on the portions that are important to understand the combination of d3 and leaflet.

Our code begins by setting up the html document in a fairly standard way.

```
<!DOCTYPE html>
<html>
<head>
        <title>d3.js with leaflet.js</title>

    <link
        rel="stylesheet"
        href="http://cdn.leafletjs.com/leaflet-0.7/leaflet.css"
    />
    <script src="http://d3js.org/d3.v3.min.js"></script>

    <script
        src="http://cdn.leafletjs.com/leaflet-0.7/leaflet.js">
    </script>

</head>
<body>

        <div id="map" style="width: 600px; height: 400px"></div>
```

[282] http://bl.ocks.org/d3noob/9267535
[283] https://gist.github.com/d3noob/9267535
[284] https://leanpub.com/D3-Tips-and-Tricks

Here we're getting some css styling and loading our leaflet.js / d3.js libraries. The only configuration item is where we set up the size of the map (in the `<div>` section and as part of the `map` div).

Then we break into the JavaScript code. The first thing we do is to project our Leaflet map;

```
var map = L.map('map').setView([-41.2858, 174.7868], 13);
mapLink =
    '<a href="http://openstreetmap.org">OpenStreetMap</a>';
L.tileLayer(
    'http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
    attribution: '&copy; ' + mapLink + ' Contributors',
    maxZoom: 18,
    }).addTo(map);
```

This is exactly the same as we have done in any of the simple map explanations in Leaflet Tips and Tricks[285] and in this case we are using the OpenStreetMap tiles.

Then we start on the d3.js part of the code.

Firstly the Leaflet map is initiated as SVG using `map._initPathRoot()`.

```
// Initialize the SVG layer
map._initPathRoot()

// We pick up the SVG from the map object
var svg = d3.select("#map").select("svg"),
g = svg.append("g");
```

Then we select the svg layer and append a `g` element to give a common reference point `g = svg.append("g")`.

Then we load the json file with the coordinates for the circles;

```
d3.json("circles.json", function(collection) {
```

Then for each of the coordinates in the `objects` section of the json data we declare a new latitude / longitude pair from the associated `coordinates`;

```
collection.objects.forEach(function(d) {
d.LatLng = new L.LatLng(d.circle.coordinates[0],
                        d.circle.coordinates[1])
})
```

Then we use a simple d3.js routine to add and place our circles based on the coordinates of each of our `objects`.

[285] https://leanpub.com/leaflet-tips-and-tricks/

```
var feature = g.selectAll("circle")
        .data(collection.objects)
        .enter().append("circle")
        .style("stroke", "black")
        .style("opacity", .6)
        .style("fill", "red")
        .attr("r", 20);
```

We declare each as a `feature` and add a bit of styling just to make them stand out.

The last 'main' part of our JavaScript makes sure that when our view of what we're looking at changes (we zoom or pan) that our d3 elements change as well;

```
map.on("viewreset", update);
update();
```

Obviously when our view changes we call the function `update`. It's the job of the `update` function to ensure that whenever the leaflet layer moves, the SVG layer with the d3.js elements follows and the points that designate the locations of those objects move appropriately;

```
function update() {
        feature.attr("transform",
        function(d) {
                return "translate("+
                        map.latLngToLayerPoint(d.LatLng).x +","+
                        map.latLngToLayerPoint(d.LatLng).y +")";
                }
        )
}
```

Here we are using the `transform` function on each `feature` to adjust the coordinates on our `LatLng` coordinates. We only need to adjust our coordinates since the size, shape, rotation and any other attribute or style is dictated by the objects themselves.

And there we have it!

d3.js circles fixed in geographic location on leaflet map but constant size

D3.js Examples Explained

I've decided to include an examples chapter because I have occasionally come across things that I wanted to do with data or a technique that didn't necessarily fall into a specific category or where I had something that I wanted to do that went beyond an exercise that I would want to turn into a simple description.

In other words I think that this will be a place where I can put random graphics that I have made up that don't necessarily fall into a logical category but which I think would be worth recording.

In many cases these examples will combine a range of techniques that have been explained in the book and in some cases they may include new material that perhaps I would have struggled to explain.

Whatever the case, I will try to explain the examples as best I can and to include a full code listing for each and a link to an electronic version wherever possible.

Dynamically retrieve historical stock records via YQL

Purpose

This page was developed to be an attempt to integrate the ability to download time range data from the Yahoo! Developer Network[286] via a YQL query and to be able to edit that query and dynamically adjust the output graph.

It doesn't hurt that the data is pretty interesting (who isn't fascinated by the rise and fall of stock prices?).

The following is a picture of the resulting graph;

Dynamic historical stock graph

The code

The following is the full code for the example. A live version is available online at bl.ocks.org[287] or GitHub[288]. It is also available as the file 'yql-dynamic-stock-line.html' as a separate download with D3 Tips and Tricks. A a copy of most the files that appear in the book can be downloaded (in a zip file) when you download the book from Leanpub[289].

[286] http://developer.yahoo.com/yql/
[287] http://bl.ocks.org/d3noob/9576689
[288] https://gist.github.com/d3noob/9576689
[289] https://leanpub.com/D3-Tips-and-Tricks

```
<!DOCTYPE html>
<meta charset="utf-8">
<style> /* set the CSS */

body { font: 12px Arial;}

path {
    stroke: steelblue;
    stroke-width: 2;
    fill: none;
}

text.shadow {
    stroke: white;
    stroke-width: 2.5px;
    opacity: 0.9;
}

.axis path,
.axis line {
    fill: none;
    stroke: grey;
    stroke-width: 1;
    shape-rendering: crispEdges;
}

</style>
<body>

<!-- set inputs for the query -->
<div id="new_input">
       
    Stock: <input type="text" name="stock" id="stock" value="YHOO"
    style="width: 70px;">
       
    Start: <input type="text" name="start" id="start" value="2013-08-10"
    style="width: 80px;">
       
    End: <input type="text" name="end" id="end" value="2014-03-10"
    style="width: 80px;">
       
    <input name="updateButton"
    type="button"
    value="Update"
    onclick="updateData()" />
</div>
```

```
<!-- load the d3.js library -->
<script src="http://d3js.org/d3.v3.min.js"></script>

<script>

// Set the dimensions of the graph
var margin = {top: 30, right: 40, bottom: 30, left: 50},
    width = 600 - margin.left - margin.right,
    height = 270 - margin.top - margin.bottom;

// Parse the date / time
var parseDate = d3.time.format("%Y-%m-%d").parse;

// Set the ranges
var x = d3.time.scale().range([0, width]);
var y = d3.scale.linear().range([height, 0]);

var xAxis = d3.svg.axis().scale(x)
    .orient("bottom").ticks(5);

var    yAxis = d3.svg.axis().scale(y)
    .orient("left").ticks(5);

var valueline = d3.svg.line()
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y(d.high); });

var svg = d3.select("body")
    .append("svg")
        .attr("width", width + margin.left + margin.right)
        .attr("height", height + margin.top + margin.bottom)
      .append("g")
        .attr("transform", "translate("
            + margin.left
            + "," + margin.top + ")");

var stock = document.getElementById('stock').value;
var start = document.getElementById('start').value;
var end = document.getElementById('end').value;

var inputURL = "http://query.yahooapis.com/v1/public/yql"+
    "?q=select%20*%20from%20yahoo.finance.historicaldata%20"+
    "where%20symbol%20%3D%20%22"
    +stock+"%22%20and%20startDate%20%3D%20%22"
    +start+"%22%20and%20endDate%20%3D%20%22"
```

```
+end+"%22&format=json&env=store%3A%2F%2F"
+"datatables.org%2Falltableswithkeys";

// Get the data
d3.json(inputURL, function(error, data){

data.query.results.quote.forEach(function(d) {
    d.date = parseDate(d.Date);
    d.high = +d.High;
    d.low = +d.Low;
});

// Scale the range of the data
x.domain(d3.extent(data.query.results.quote, function(d) {
    return d.date; }));
y.domain([
    d3.min(data.query.results.quote, function(d) { return d.low; }),
    d3.max(data.query.results.quote, function(d) { return d.high; })
]);

svg.append("path")        // Add the valueline path.
    .attr("class", "line")
    .attr("d", valueline(data.query.results.quote));

svg.append("g")            // Add the X Axis
    .attr("class", "x axis")
    .attr("transform", "translate(0," + height + ")")
    .call(xAxis);

svg.append("g")            // Add the Y Axis
    .attr("class", "y axis")
    .call(yAxis);

svg.append("text")          // Add the label
    .attr("class", "label")
    .attr("transform", "translate(" + (width+3) + ","
        + y(data.query.results.quote[0].high) + ")")
    .attr("dy", ".35em")
    .attr("text-anchor", "start")
    .style("fill", "steelblue")
    .text("high");

svg.append("text")          // Add the title shadow
    .attr("x", (width / 2))
    .attr("y", margin.top / 2)
    .attr("text-anchor", "middle")
```

```
        .attr("class", "shadow")
        .style("font-size", "16px")
        .text(stock);

    svg.append("text")          // Add the title
        .attr("class", "stock")
        .attr("x", (width / 2))
        .attr("y", margin.top / 2)
        .attr("text-anchor", "middle")
        .style("font-size", "16px")
        .text(stock);
});

// ** Update data section (Called from the onclick)
function updateData() {

var stock = document.getElementById('stock').value;
var start = document.getElementById('start').value;
var end = document.getElementById('end').value;

var inputURL = "http://query.yahooapis.com/v1/public/yql"+
    "?q=select%20*%20from%20yahoo.finance.historicaldata%20"+
    "where%20symbol%20%3D%20%22"
    +stock+"%22%20and%20startDate%20%3D%20%22"
    +start+"%22%20and%20endDate%20%3D%20%22"
    +end+"%22&format=json&env=store%3A%2F%2F"
    +"datatables.org%2Falltableswithkeys";

    // Get the data again
    d3.json(inputURL, function(error, data){

        data.query.results.quote.forEach(function(d) {
            d.date = parseDate(d.Date);
            d.high = +d.High;
            d.low = +d.Low;
        });

        // Scale the range of the data
        x.domain(d3.extent(data.query.results.quote, function(d) {
            return d.date; }));
        y.domain([
            d3.min(data.query.results.quote, function(d) {
                return d.low; }),
            d3.max(data.query.results.quote, function(d) {
                return d.high; })
        ]);
```

```
        // Select the section we want to apply our changes to
        var svg = d3.select("body").transition();

        // Make the changes
        svg.select(".line")    // change the line
            .duration(750)
            .attr("d", valueline(data.query.results.quote));

        svg.select(".label")   // change the label text
            .duration(750)
            .attr("transform", "translate(" + (width+3) + ","
            + y(data.query.results.quote[0].high) + ")");

        svg.select(".shadow") // change the title shadow
            .duration(750)
            .text(stock);

        svg.select(".stock")   // change the title
            .duration(750)
            .text(stock);

        svg.select(".x.axis") // change the x axis
            .duration(750)
            .call(xAxis);
        svg.select(".y.axis") // change the y axis
            .duration(750)
            .call(yAxis);
    });
}

</script>
</body>
```

The description

Firstly, I have not included any form of validation or sanitising of the input fields. If you were to build something that was being used in a serious way, that would be essential.

Secondly, there are limits on what the YQL query will return. I have found that there appears to be a limit on the date range allowed (although I'm not sure what that limit is) and there is of course a limit to what the Yahoo! Developer Network will support for different end use cases. If you want to use the data for commercial reasons or if your use is heavy[290], you will need to contact them to arrange for some form of agreement to use the data appropriately.

[290] http://developer.yahoo.com/yql/guide/usage_info_limits.html

To use the graph all you need to do is enter a valid ticker symbol[291] and a start / end date range where the date is formatted as yyyy/mm/dd. As I noted earlier, there appears to be a range limit, so feel free to experiment a bit to work it out if necessary to your use.

Stock: YHOO Start: 2013-08-10 End: 2014-03-10 Update

The ticker symbol and start/stop dates

The section to get the input fields was something new to me as normally I would use bootstrap.js[292] with it's wealth of form input options. But the following section in the HTLM portion was neat enough to get the required input.

```
<div id="new_input">
       
    Stock: <input type="text" name="stock" id="stock" value="YHOO"
    style="width: 70px;">
       
    Start: <input type="text" name="start" id="start" value="2013-08-10"
    style="width: 80px;">
       
    End: <input type="text" name="end" id="end" value="2014-03-10"
    style="width: 80px;">
       
    <input name="updateButton"
    type="button"
    value="Update"
    onclick="updateData()" />
</div>
```

Of course it needs to be coupled with a JavaScript section to allow it to use the inputted fields in the query but that was also nice and easy with the following section of code;

```
var stock = document.getElementById('stock').value;
var start = document.getElementById('start').value;
var end = document.getElementById('end').value;
```

The HTML portion includes the `onclick="updateData()"` code that allows the JavaScript `updateData` function to be called that reloads new data from the Yahoo! Developer Network and updates the d3.js objects.

This particular file uses the 'load everything first' then 'update everything that needs updating' model that was followed in the earlier chapter on creating a line graph that loads data dynamically.

The YQL query is declared as a variable in the following section;

[291] http://en.wikipedia.org/wiki/Ticker_symbol
[292] http://getbootstrap.com/

```
var inputURL = "http://query.yahooapis.com/v1/public/yql"+
    "?q=select%20*%20from%20yahoo.finance.historicaldata%20"+
    "where%20symbol%20%3D%20%22"
    +stock+"%22%20and%20startDate%20%3D%20%22"
    +start+"%22%20and%20endDate%20%3D%20%22"
    +end+"%22&format=json&env=store%3A%2F%2F"
    +"datatables.org%2Falltableswithkeys";
```

It has had line feeds deliberately introduced to make formatting on the pages of the book easier (otherwise the publishing process introduces additional characters). In it you can see the addition of the variables that allow the query to be executed (`stock`, `start` and `end`).

Immediately after loading the data we run it through a `forEach` loop that goes to the location in the JSON hierarchy where the `High`, `Low` and `Date` values are stored and it ensures that the `high` and `low` values are correctly recognises as numbers and formats the date.

```
data.query.results.quote.forEach(function(d) {
    d.date = parseDate(d.Date);
    d.high = +d.High;
    d.low = +d.Low;
});
```

This is quite interesting because it provides a peek at the structure of the JSON. This is a pretty important piece of information because without the structure, it is not possible to correctly address the data you want. I'm not sure what the best method would be for determining the structure of the returned data, but I simply use a `console.log(data)` call after the data is loaded while I am developing the file and this allows me to explore and note the structure.

The following screen-shot illustrates the method;

console.log(data) structure

You should be able to discern the `.query.results.quote` pathway that leads to the `High`, `Low` and `Date` values.

The remainder of the code is a repetition of examples explained in the remainder of the book. Most especially in the simple line graph area.

Linux Processes via Interactive Tree diagram

Purpose

This page was developed to play with the idea of visualizing the relationship between processes running on a Linux server. To my shame, I never twigged that since the processes were ordered in a hierarchy they would therefore make an excellent tree diagram. I am therefore indebted to a friend for pointing out the obvious (as he often needs to do :-)).

Ultimately I have grand visions of this type of display being used to illustrate excessive memory or CPU usage when fault conditions occur, but for the purposes of simply showing the relationships, this example is suitable.

There are obviously a lot of processes running on a Linux server, so it was necessary to make the diagram interactive to allow branches to collapse where required for clarity. Indeed, there are a few of the tree diagram features which are covered separately in the chapter on tree diagrams which are combined here (interactive nodes, loading from an external source and making the tree interactive). Additionally, there is a great deal of data about each node that is available when running the `ps` command. I have chosen to show some of these in a tool tip that will appear when hovered over a node.

The tree is fairly large, so the following is a section of the tree with tool tip in action

Linux processes in tree form

The Code

The following is the full code for the example. A live version is available online at bl.ocks.org[293] or GitHub[294]. It is also available as the files 'process-tree.html' and 'ps.csv' as separate downloads with the book D3 Tips and Tricks. A a copy of most the files that appear in the book can be downloaded (in a zip file) when you download the book from Leanpub[295].

[293] http://bl.ocks.org/d3noob/9692795
[294] https://gist.github.com/d3noob/9692795
[295] https://leanpub.com/D3-Tips-and-Tricks

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">

    <title>Linux Process Tree</title>

    <style>

  div.tooltip {
    position: absolute;
    text-align: left;
    width: 180px;
    height: 80px;
    padding: 2px;
    font: 12px sans-serif;
    background: lightsteelblue;
    border: 0px;
    border-radius: 8px;
    pointer-events: none;
  }

  .node circle {
    fill: #fff;
    stroke: steelblue;
    stroke-width: 3px;
  }

  .node text { font: 12px sans-serif; }

  .link {
    fill: none;
    stroke: #ccc;
    stroke-width: 2px;
  }

    </style>

  </head>

  <body>

<!-- load the d3.js library -->
<script src="http://d3js.org/d3.v3.min.js"></script>

<script>
```

```
// ************** Generate the tree diagram   *****************
var margin = {top: 20, right: 120, bottom: 20, left: 120},
  width = 1200 - margin.right - margin.left,
  height = 900 - margin.top - margin.bottom;

var i = 0;
  duration = 750;

var tree = d3.layout.tree()
  .size([height, width]);

var diagonal = d3.svg.diagonal()
  .projection(function(d) { return [d.y, d.x]; });

var svg = d3.select("body").append("svg")
  .attr("width", width + margin.right + margin.left)
  .attr("height", height + margin.top + margin.bottom)
  .append("g")
  .attr("transform", "translate(" + margin.left + "," +
                                    margin.top + ")");

// load the external data
d3.csv("ps.csv", function(error, data) {

  // *********** Convert flat data into a nice tree ***************
  // create a name: node map
  var dataMap = data.reduce(function(map, node) {
    map[node.name] = node;
    return map;
  }, {});

  // create the tree array
  var treeData = [];
  data.forEach(function(node) {
    // add to parent
    var parent = dataMap[node.parent];
    if (parent) {
      // create child array if it doesn't exist
      (parent.children || (parent.children = []))
        // add node to child array
        .push(node);
    } else {
      // parent is null or missing
      treeData.push(node);
    }
```

```
  });

  root = treeData[0];
  root.x0 = height / 2;
  root.y0 = 0;

  update(root);
});

d3.select(self.frameElement).style("height", "500px");

function update(source) {

  // Compute the new tree layout.
  var nodes = tree.nodes(root).reverse(),
    links = tree.links(nodes);

  // Normalize for fixed-depth.
  nodes.forEach(function(d) { d.y = d.depth * 180; });

  // Update the nodesâ€¦
  var node = svg.selectAll("g.node")
    .data(nodes, function(d) { return d.id || (d.id = ++i); });

  // Enter any new nodes at the parent's previous position.
  var nodeEnter = node.enter().append("g")
    .attr("class", "node")
    .attr("transform", function(d) {
      return "translate(" + source.y0 + "," + source.x0 + ")"; })
    .on("click", click)
      // add tool tip for ps -eo pid,ppid,pcpu,size,comm,ruser,s
      .on("mouseover", function(d) {
        div.transition()
          .duration(200)
          .style("opacity", .9);
        div .html(
            "PID: " + d.name + "<br/>" +
            "Command: " + d.COMMAND + "<br/>" +
            "User: " + d.RUSER + "<br/>" +
            "%CPU: " + d.CPU + "<br/>" +
            "Memory: " + d.SIZE
            )
          .style("left", (d3.event.pageX) + "px")
          .style("top", (d3.event.pageY - 28) + "px");
        })
      .on("mouseout", function(d) {
```

```
        div.transition()
          .duration(500)
          .style("opacity", 0);
        });

  nodeEnter.append("circle")
    .attr("r", 1e-6)
    .style("fill", function(d) {
      return d._children ? "lightsteelblue" : "#fff"; });

  nodeEnter.append("text")
    .attr("x", function(d) {
      return d.children || d._children ? -13 : 13; })
    .attr("dy", ".35em")
    .attr("text-anchor", function(d) {
      return d.children || d._children ? "end" : "start"; })
    .text(function(d) { return d.COMMAND; })
    .style("fill-opacity", 1e-6);

// add the tool tip
  var div = d3.select("body").append("div")
    .attr("class", "tooltip")
    .style("opacity", 0);

  // Transition nodes to their new position.
  var nodeUpdate = node.transition()
    .duration(duration)
    .attr("transform", function(d) {
        return "translate(" + d.y + "," + d.x + ")";
      });

  nodeUpdate.select("circle")
    .attr("r", 10)
    .style("fill", function(d) {
          return d._children ? "lightsteelblue" : "#fff"; });

  nodeUpdate.select("text")
    .style("fill-opacity", 1);

  // Transition exiting nodes to the parent's new position.
  var nodeExit = node.exit().transition()
    .duration(duration)
    .attr("transform", function(d) { return "translate(" + source.y +
                                            "," + source.x + ")"; })
    .remove();
```

```
  nodeExit.select("circle")
    .attr("r", 1e-6);

  nodeExit.select("text")
    .style("fill-opacity", 1e-6);

  // Update the linksâ€¦
  var link = svg.selectAll("path.link")
    .data(links, function(d) { return d.target.id; });

  // Enter any new links at the parent's previous position.
  link.enter().insert("path", "g")
    .attr("class", "link")
    .attr("d", function(d) {
      var o = {x: source.x0, y: source.y0};
      return diagonal({source: o, target: o});
    });

  // Transition links to their new position.
  link.transition()
    .duration(duration)
    .attr("d", diagonal);

  // Transition exiting nodes to the parent's new position.
  link.exit().transition()
    .duration(duration)
    .attr("d", function(d) {
      var o = {x: source.x, y: source.y};
      return diagonal({source: o, target: o});
    })
    .remove();

  // Stash the old positions for transition.
  nodes.forEach(function(d) {
  d.x0 = d.x;
  d.y0 = d.y;
  });
}

// Toggle children on click.
function click(d) {
  if (d.children) {
    d._children = d.children;
    d.children = null;
  } else {
    d.children = d._children;
```

```
    d._children = null;
  }
  update(d);
}

</script>

  </body>
</html>
```

Description

I will describe both the code for the example and the csv file that accompanies it since (in this case) the data that generates the tree is not gathered entirely automatically and has some manual intervention applied to make it suitable for purpose.

The csv file (`ps.csv`) was generated by running the command...

```
ps -eo pid,ppid,pcpu,size,comm,ruser,s
```

...and converting the resultant output to a csv file.

```
name,parent,CPU,SIZE,COMMAND,RUSER,S
0, ,0,0,start,nul,u
1,0,0.0,1140,init,root,S
2,0,0.0,0,kthreadd,root,S
3,2,0.0,0,ksoftirqd/0,root,S
5,2,0.0,0,kworker/0:0H,root,S
6,2,0.0,0,kworker/u2:0,root,S
7,2,0.0,0,migration/0,root,S
8,2,0.0,0,rcu_bh,root,S
9,2,0.0,0,rcuob/0,root,S
10,2,0.0,0,rcu_sched,root,S
...
```

The column names I have specifically asked for with the command are;

- pid: Process ID - The unique numeric identifier assigned to the process
- ppid: Parent Process ID - Indicates the decimal value of the parent process ID
- pcpu: Percentage of CPU - Time used (total CPU time divided by length of time the process has been running)
- size: Size - Memory size in kilobytes
- comm: Command -Indicates the short name of the command being executed
- ruser: Real User ID - The textual user ID
- s: State - Process state with possible values:

- R Running
- S Sleeping (may be interrupted)
- D Sleeping (may not be interrupted) used to indicate process is handling input/output
- T Stopped or being traced
- Z Zombie or "hung" process

I manually added the 'start' line (0, ,0,0,start,nul,u) to include the root node and removed the '%' sign from the '%CPU' label (which is produces when running 'ps') to reduce chance of errors.

In theory the process of formatting the data file could be automated (indeed, there may be a much better way to gather and include it!).

The code is essentially an amalgam of four components which have been covered separately in earlier sections of the book;

1. Tree code where the data is loaded from an external source
2. Tree code where the data is converted into a hierarchy from a flat file
3. Tree code to allow the diagram to collapse and expand.
4. Tool tips as an HTML object.

The loading of the data from an external source occurs in this portion of the code;

```
// load the external data
d3.csv("ps.csv", function(error, data) {

  // *********** Convert flat data into a nice tree ***************
  // create a name: node map
  var dataMap = data.reduce(function(map, node) {
    map[node.name] = node;
    return map;
  }, {});

  // create the tree array
  var treeData = [];
  data.forEach(function(node) {
    // add to parent
    var parent = dataMap[node.parent];
    if (parent) {
      // create child array if it doesn't exist
      (parent.children || (parent.children = []))
        // add node to child array
        .push(node);
    } else {
      // parent is null or missing
      treeData.push(node);
    }
  });
```

```
  root = treeData[0];
  root.x0 = height / 2;
  root.y0 = 0;

  update(root);
});
```

In reality the loading process is just the wrapping part of that code segment as the inner portion is the section that takes the flat `data` and creates the hierarchical `treeData`.

The function `update` is the main section that allows the tree to expand and collapse (along with the `click` function). I won't repeat that code here as it is quite lengthy and probably unnecessary (as it appears only a few pages previously). The only difference between the `update` function here and the one that is used in the example in the tree chapter is that we include a portion of code that allows us to include some of the details of each process in a tool tip;

```
  // Enter any new nodes at the parent's previous position.
  var nodeEnter = node.enter().append("g")
    .attr("class", "node")
    .attr("transform", function(d) {
      return "translate(" + source.y0 + "," + source.x0 + ")"; })
    .on("click", click)
      // add tool tip for ps -eo pid,ppid,pcpu,size,comm,ruser,s
      .on("mouseover", function(d) {
        div.transition()
          .duration(200)
          .style("opacity", .9);
        div.html(
            "PID: " + d.name + "<br/>" +
            "Command: " + d.COMMAND + "<br/>" +
            "User: " + d.RUSER + "<br/>" +
            "%CPU: " + d.CPU + "<br/>" +
            "Memory: " + d.SIZE
            )
          .style("left", (d3.event.pageX) + "px")
          .style("top", (d3.event.pageY - 28) + "px");
        })
      .on("mouseout", function(d) {
        div.transition()
          .duration(500)
          .style("opacity", 0);
        });
```

We use the `mouseover` and `mouseout` calls to find out when the cursor is over a portion of a node (and this includes the text part as well as the circle) and print out the name of the process (which

is the Process ID (PID)), the command name (a nice textual equivalent of the PID) the user that the process is run by along with the CPU use and memory used.

The tree is pretty large, so depending on your use it might want to expand or perhaps contract. It could be drawn radially perhaps and there is certainly scope for encoding information about memory and CPU usage in the associated colouring of the nodes / links.

I would recommend visiting the demo page on bl.ocks.org[296] to get a good look at the result, as no picture in a book will be able to capture it sufficiently :-)

Linux processes in tree form

[296] http://bl.ocks.org/d3noob/9692795

Multi-line graph with automatic legend and toggling show / hide lines.

Purpose

Creating a multi-line graph is a pretty handy thing to be able to do and we worked through an example earlier in the book as an extension of our simple graph. In that example we used a csv file that had the data arranged with each lines values in a separate column.

```
date,close,open
1-May-12,68.13,34.12
30-Apr-12,63.98,45.56
27-Apr-12,67.00,67.89
26-Apr-12,89.70,78.54
25-Apr-12,99.00,89.23
24-Apr-12,130.28,99.23
23-Apr-12,166.70,101.34
```

This is a common way to have data stored, but if you are retrieving information from a database, you may not have the luxury of having it laid out in columns. It may be presented in a more linear fashion where each lines values are stores on a unique row with the identifier for the line on the same row. For instance, the data above could just as easily be presented as follows;

```
price,date,value
close,1-May-12,68.13
close,30-Apr-12,63.98
close,27-Apr-12,67.00
close,26-Apr-12,89.70
close,25-Apr-12,99.00
close,24-Apr-12,130.28
close,23-Apr-12,166.70
open,1-May-12,34.12
open,30-Apr-12,45.56
open,27-Apr-12,67.89
open,26-Apr-12,78.54
open,25-Apr-12,89.23
open,24-Apr-12,99.23
open,23-Apr-12,101.34
```

In this case, we would need to 'pivot' the data to produce the same multi-column representation as the original format. This is not always easy, but it can be achieved using the d3 `nest` function which we will examine.

As well as this we will want to automatically encode the lines to make them different colours and to add a legend with the line name and the colour of the appropriate line.

Finally, because we will build a graph script that can cope with any number of lines (within reason), we will need to be able to show / hide the individual lines to try and clarify the graph if it gets too cluttered.

All of these features have been covered individually in the book, so what we're going to do is combine them in a way that presents us with an elegant multi-line graph that looks a bit like this;

Multi-line graph with legend

The Code

The following is the code for the initial example which is a slight derivative of the original simple graph. A live version is available online at bl.ocks.org[297] or GitHub[298]. It is also available as the files 'super-multi-lines.html' and 'stocks.csv' as a download with the book D3 Tips and Tricks (in a zip file) when you download the book from Leanpub[299].

```
<!DOCTYPE html>
<meta charset="utf-8">
<style> /* set the CSS */

body { font: 12px Arial;}

path {
    stroke: steelblue;
    stroke-width: 2;
    fill: none;
}

.axis path,
.axis line {
    fill: none;
    stroke: grey;
    stroke-width: 1;
    shape-rendering: crispEdges;
}

</style>
<body>

<!-- load the d3.js library -->
<script src="http://d3js.org/d3.v3.min.js"></script>
```

[297] http://bl.ocks.org/d3noob/d8be922a10cb0b148cd5
[298] https://gist.github.com/d3noob/d8be922a10cb0b148cd5
[299] https://leanpub.com/D3-Tips-and-Tricks

```
<script>

// Set the dimensions of the canvas / graph
var margin = {top: 30, right: 20, bottom: 30, left: 50},
    width = 600 - margin.left - margin.right,
    height = 270 - margin.top - margin.bottom;

// Parse the date / time
var parseDate = d3.time.format("%b %Y").parse;

// Set the ranges
var x = d3.time.scale().range([0, width]);
var y = d3.scale.linear().range([height, 0]);

// Define the axes
var xAxis = d3.svg.axis().scale(x)
    .orient("bottom").ticks(5);

var yAxis = d3.svg.axis().scale(y)
    .orient("left").ticks(5);

// Define the line
var priceline = d3.svg.line()
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y(d.price); });

// Adds the svg canvas
var svg = d3.select("body")
    .append("svg")
        .attr("width", width + margin.left + margin.right)
        .attr("height", height + margin.top + margin.bottom)
    .append("g")
        .attr("transform",
              "translate(" + margin.left + "," + margin.top + ")");

// Get the data
d3.csv("stocks.csv", function(error, data) {
    data.forEach(function(d) {
                d.date = parseDate(d.date);
                d.price = +d.price;
    });

    // Scale the range of the data
    x.domain(d3.extent(data, function(d) { return d.date; }));
    y.domain([0, d3.max(data, function(d) { return d.price; })]);
```

```
    // Nest the entries by symbol
    var dataNest = d3.nest()
        .key(function(d) {return d.symbol;})
        .entries(data);

    // Loop through each symbol / key
    dataNest.forEach(function(d) {

        svg.append("path")
            .attr("class", "line")
            .attr("d", priceline(d.values));

    });

    // Add the X Axis
    svg.append("g")
        .attr("class", "x axis")
        .attr("transform", "translate(0," + height + ")")
        .call(xAxis);

    // Add the Y Axis
    svg.append("g")
        .attr("class", "y axis")
        .call(yAxis);

});

</script>
</body>
```

Description

Nesting the data

The example code above differs from the simple graph in two main ways.

Firstly, the script loads the file `stocks.csv` which was used by Mike Bostock in his small multiples example[300]. This means that the variable names used are different (`price` for the value of the stocks, `symbol` for the name of the stock and good old `date` for the date) and we have to adjust the `parseDate` function to parse a modified date value.

Secondly we add the code blocks to take the `stocks.csv` information that we load as `data` and we apply the `d3.nest` function to it and draw each line.

The following code nest's the data

[300] http://bl.ocks.org/mbostock/1157787

```
var dataNest = d3.nest()
    .key(function(d) {return d.symbol;})
    .entries(data);
```

We declare our new array's name as `dataNest` and we initiate the `nest` function;

```
var dataNest = d3.nest()
```

We assign the `key` for our new array as `symbol`. A 'key' is like a way of saying "*This is the thing we will be grouping on*". In other words our resultant array will have a single entry for each unique `symbol` or stock which will itself be an array of dates and values.

```
.key(function(d) {return d.symbol;})
```

Then we tell the nest function which data array we will be using for our source of data.

```
}).entries(data);
```

Then we use the nested data to loop through our stocks and draw the lines

```
dataNest.forEach(function(d) {

    svg.append("path")
        .attr("class", "line")
        .attr("d", priceline(d.values));

});
```

The `forEach` function being applied to `dataNest` means that it will take each of the keys that we have just declared with the `d3.nest` (each stock) and use the values for each stock to append a line using its values.

The end result looks like the following;

A very plain multi-line graph

You would be justified in thinking that this is more than a little confusing. Clearly while we have been successful in making each stock draw a corresponding line, unless we can tell them apart, the graph is pretty useless.

Applying the colours

Making sure that the colours that are applied to our lines (and ultimately our legend text) is unique from line to line is actually pretty easy.

The code that we will implement for this change is available online at bl.ocks.org[301] or GitHub[302]. It is also available as the files 'super-multi-colours.html' and 'stocks.csv' as a download with the book D3 Tips and Tricks (in a zip file) when you download the book from Leanpub[303].

The changes that we will make to our code are captured in the following code snippet.

```
var color = d3.scale.category10();

// Loop through each symbol / key
dataNest.forEach(function(d) {

    svg.append("path")
        .attr("class", "line")
        .style("stroke", function() {
            return d.color = color(d.key); })
        .attr("d", priceline(d.values));

});
```

Firstly we need to declare an ordinal scale[304] for our colours with `var color = d3.scale.category10();`. This is a set of categorical colours (10 of them in this case) that can be invoked which are a nice mix of difference from each other and pleasant on the eye.

We then use the colour scale to assign a unique stroke (line colour) for each unique key (symbol) in our dataset.

```
.style("stroke", function() {
    return d.color = color(d.key); })
```

It seems easy when it's implemented, but in all reality, it is the product of some very clever thinking behind the scenes when designing d3.js and even picking the colours that are used. The end result is a far more usable graph of the stock prices.

[301] http://bl.ocks.org/d3noob/88d2a87b72ea894c285c
[302] https://gist.github.com/d3noob/88d2a87b72ea894c285c
[303] https://leanpub.com/D3-Tips-and-Tricks
[304] https://github.com/mbostock/d3/wiki/Ordinal-Scales

Multi-line graph with unique colours

Of course now we're faced with the problem of not knowing which line represents which stock price. Time for a legend.

Adding the legend

If we think about the process of adding a legend to our graph, what we're trying to achieve is to take every unique data series we have (stock) and add a relevant label showing which colour relates to which stock. At the same time, we need to arrange the labels in such a way that they are presented in a manner that is not offensive to the eye. In the example I will go through I have chosen to arrange them neatly spaced along the bottom of the graph. so that the final result looks like the following;

Multi-line graph with legend

Bear in mind that the end result will align the legend completely automatically. If there are three stocks it will be equally spaced, if it is six stocks they will be equally spaced. The following is a reasonable mechanism to facilitate this, but if the labels for the data values are of radically different lengths, the final result will looks 'odd' likewise, if there are a LOT of data values, the legend will start to get crowded.

The code that we will implement for this change is available online at bl.ocks.org[305] or GitHub[306]. It is also available as the files 'super-multi-legend.html' and 'stocks.csv' as a download with the book D3 Tips and Tricks (in a zip file) when you download the book from Leanpub[307].

There are three broad categories of changes that we will want to make to our current code;

1. Declare a style for the legend font
2. Change the area and margins for the graph to accommodate the additional text
3. Add the text

Declaring the style for the legend text is as easy as making an appropriate entry in the `<style>` section of the code. For this example I have chosen the following;

[305] http://bl.ocks.org/d3noob/7cd5a74c4620db72f43f
[306] https://gist.github.com/d3noob/7cd5a74c4620db72f43f
[307] https://leanpub.com/D3-Tips-and-Tricks

```
.legend {
    font-size: 16px;
    font-weight: bold;
    text-anchor: middle;
}
```

To change the area and margins of the graph we can make the following small changes to the code.

```
var margin = {top: 30, right: 20, bottom: 70, left: 50},
    width = 600 - margin.left - margin.right,
    height = 300 - margin.top - margin.bottom;
```

The bottom margin is now 70 pixels high and the overall space for the area that the graph (including the margins) covers is increased to 300 pixels.

To add the legend text is slightly more work, but only slightly more. The following code incorporates the changes and I have placed commented out asterisks to the end of the lines that have been added

```
    legendSpace = width/dataNest.length; // spacing for legend // ******

    // Loop through each symbol / key
    dataNest.forEach(function(d,i) {                          // ******

        svg.append("path")
            .attr("class", "line")
            .style("stroke", function() { // Add the colours dynamically
                return d.color = color(d.key); })
            .attr("d", priceline(d.values));

        // Add the Legend
        svg.append("text")                                   // *******
            .attr("x", (legendSpace/2)+i*legendSpace) // spacing // ****
            .attr("y", height + (margin.bottom/2)+ 5)        // *******
            .attr("class", "legend")    // style the legend   // *******
            .style("fill", function() { // dynamic colours    // *******
                return d.color = color(d.key); })             // *******
            .text(d.key);                                     // *******

    });
```

The first added line finds the spacing between each legend label by dividing the width of the graph area by the number of symbols (`key`'s or stocks).

```
legendSpace = width/dataNest.length;
```

Then there is a small and subtle change that might other wise go unnoticed, but is nonetheless significant. We add an `i` to the `forEach` function;

```
dataNest.forEach(function(d,i) {
```

This might not seem like much of a big deal, but declaring `i` allows us to access the index of the returned data. This means that each unique `key` (stock or symbol) has a unique number. In our example those numbers would be from 0 to 3 (MSFT = 0, AMZN = 1, IBM = 2 and AAPL = 3 (this is the order in which the stocks appear in our csv file)).

Now we get to adding our text. Again, this is a fairly simple exercise which is following the route that we have taken several times already in the book but using some of our prepared values.

```
svg.append("text")
    .attr("x", (legendSpace/2)+i*legendSpace)
    .attr("y", height + (margin.bottom/2)+ 5)
    .attr("class", "legend")
    .style("fill", function() {
        return d.color = color(d.key); })
    .text(d.key);
```

The horizontal spacing for the labels is achieved by setting each label to the position set by the index associated with the label and the space available on the graph. To make it work out nicely we add half a `legendSpace` at the start (`legendSpace/2`) and then add the product of the index (`i`) and `legendSpace` (`i*legendSpace`).

We position the legend vertically so that it is in the middle of the bottom margin (`height + (margin.bottom/2)+ 5`).

And we apply the same colour function to the text as we did to the lines earlier;

```
    .style("fill", function() {
        return d.color = color(d.key); })
```

The final result is a neat and tidy legend at the bottom of the graph;

Multi-line graph with legend

If you're looking for an exercise to test your skills you could adapt the code to show the legend to the right of the graph. And if you wanted to go one better, you could arrange the order of the legend to reflect the final numeric value on the right of the graph (I.e in this case AAPL would be on the top and MSFT on the bottom).

Making it interactive

The last step we'll take in this example is to provide ourselves with a bit of control over how the graph looks. Even with the multiple colours, the graph could still be said to be 'busy'. To clean it up or at least to provide the ability to more clearly display the data that a user wants to see we will add code that will allow us to click on a legend label and this will toggle the corresponding graph line on or off.

This is a progression from the example of how to show / hide an element by clicking on another element that was introduced in he 'Assorted tips and tricks' chapter.

The only changes to our code that need to be implemented are in the `forEach` section below. I have left some comments with asterisks in the code below to illustrate lines that are added.

```
dataNest.forEach(function(d,i) {

    svg.append("path")
        .attr("class", "line")
        .style("stroke", function() {
            return d.color = color(d.key); })
        .attr("id", 'tag'+d.key.replace(/\s+/g, '')) // assign ID **
        .attr("d", priceline(d.values));

    // Add the Legend
    svg.append("text")
        .attr("x", (legendSpace/2)+i*legendSpace)
        .attr("y", height + (margin.bottom/2)+ 5)
        .attr("class", "legend")
        .style("fill", function() {
            return d.color = color(d.key); })
        .on("click", function(){                        // ************
            // Determine if current line is visible
            var active   = d.active ? false : true,     // ************
            newOpacity = active ? 0 : 1;                // ************
            // Hide or show the elements based on the ID
            d3.select("#tag"+d.key.replace(/\s+/g, '')) // *********
                .transition().duration(100)             // ************
                .style("opacity", newOpacity);          // ************
            // Update whether or not the elements are active
            d.active = active;                          // ************
            })                                          // ************
        .text(d.key);

});
```

The full code for the complete working example is available online at bl.ocks.org[308] or GitHub[309].

[308] http://bl.ocks.org/d3noob/e99a762017060ce81c76
[309] https://gist.github.com/d3noob/e99a762017060ce81c76

It is also available as the files 'super-multi.html' and 'stocks.csv' as a download with the book D3 Tips and Tricks (in a zip file) when you download the book from Leanpub[310].

The first piece of code that we need to add assign an `id` to each legend text label.

```
.attr("id", 'tag'+d.key.replace(/\s+/g, ''))
```

Being able to use our `key` value as the id means that each label will have a unique identifier. "*What's with adding the `'tag'` piece of text to the `id`?*" I hear you ask. Good question. If our key starts with a number we could strike trouble (in fact I'm sure there are plenty of other ways we could strike trouble too, but this was one I came accross). As well as that we include a little regular expression goodness to strip any spaces out of the key with `.replace(/\s+/g, '')`.

The `.replace` calls the regular expression action on our `key`. `\s` is the regex for "whitespace", and `g` is the "global" flag, meaning match ALL `\s` (whitespaces). The `+` allows for any *contiguous* string of space characters to being replaced with the empty string (`''`). This was a late addition to the example and kudos go to the participants in the Stack Overflow question here[311].

Then we use the `.on("click", function(){` call carry out some actions on the label if it is clicked on. We toggle the `.active` descriptor for our element with `var active = d.active ? false : true,`. Then we set the value of `newOpacity` to either `0` or `1` depending on whether `active` is false or true.

From here we can select our label using its unique id and adjust it's opacity to either `0` (transparent) or `1` (opaque);

```
d3.select("#tag"+d.key.replace(/\s+/g, ''))
    .transition().duration(100)
    .style("opacity", newOpacity);
```

Just because we can, we also add in a `transition` statement so that the change in transparency doesn't occur in a flash (100 milli seconds in fact (`.duration(100)`)).

Lastly we update our `d.active` variable to whatever the active state is so that it can toggle correctly the next time it is clicked on.

Since it's kind of difficult to represent interactivity in a book, head on over to the live example on bl.ocks.org[312] to see the toggling awesomeness that could be yours!

[310] https://leanpub.com/D3-Tips-and-Tricks
[311] http://stackoverflow.com/questions/5963182/how-to-remove-spaces-from-a-string-using-javascript
[312] http://bl.ocks.org/d3noob/e99a762017060ce81c76

My Favourite tooltip method for a line graph.

Purpose

Tooltips are a fabulous way to include an interactive element on a graph and a great mechanism for including additional, focussed information to the user.

There are quite a number of different ways to implement tooltips (one of which you can find in the 'Adding tooltips' section of the 'Assorted Tips and Tricks' chapter of D3 Tips and Tricks) and I would be very hesitant about proclaiming any one better than another. However, the one we will work through here is my favourite when using a line graph as I think it brings a 'fuzzier' mechanism for deciding when a tooltip is highlighted (you don't have to be over an object to get information on it) which I like.

I believe that the original example for this was shown by Mike Bostock here[313], but I first came across the technique in an example by 'gniemetz'[314]. I liked 'gniemetz's example enough to adapt a similar example which I will explain below.

The idea with this technique is to set an area the size of the graph that will be used to determine when a tooltip will be displayed. So that when the mouse enters that area, the `display` style that allows elements to be shown or hidden. This then tells the script to show the tooltip and the location of the mouse determines which point will have the tooltip. In the example below we can see that the mouse cursor is some distance away from the point that is being highlighted, but it is in line (in the vertical axis) with the highlighted point (in fact we will use some clever maths to determine which date point (or point on the x axis) is the one that will be used to generate the tooltip.

More complicated favourite tooltip example

To begin this explanation we'll start with a simple example that will just project a circle on the point where the tooltip will appear. Once we've worked out how that works we can add whatever we want and I will explain what is going on in the more complex example.

As mentioned, we will start with a simple example that adds a circle on the point where we will place our tooltip. It will look a bit like this;

[313] http://bl.ocks.org/mbostock/3902569
[314] http://bl.ocks.org/gniemetz/4618602

Simple version of the favourite tooltip example

The Code

The full code for this simple example is available online at bl.ocks.org[315] or GitHub[316]. It is also available as the files 'best-tooltip-simple.html' and 'atad.csv' as a download with the book D3 Tips and Tricks (in a zip file) when you download the book from Leanpub[317].

I have placed commented out asterisks besides the lines that have been added or altered from the simple graph example that we started out with at the beginning of the book so that it's easy to see what has changed.

```
<!DOCTYPE html>
<meta charset="utf-8">
<style> /* set the CSS */

body { font: 12px Arial;}

path {
    stroke: steelblue;
    stroke-width: 2;
    fill: none;
}

.axis path,
.axis line {
    fill: none;
    stroke: grey;
    stroke-width: 1;
    shape-rendering: crispEdges;
}

</style>
```

[315] http://bl.ocks.org/d3noob/e5daff57a04c2639125e
[316] https://gist.github.com/d3noob/e5daff57a04c2639125e
[317] https://leanpub.com/D3-Tips-and-Tricks

```
<body>

<!-- load the d3.js library -->
<script src="http://d3js.org/d3.v3.min.js"></script>

<script>

// Set the dimensions of the canvas / graph
var margin = {top: 30, right: 20, bottom: 30, left: 50},
    width = 600 - margin.left - margin.right,
    height = 270 - margin.top - margin.bottom;

// Parse the date / time
var parseDate = d3.time.format("%d-%b-%y").parse;
    bisectDate = d3.bisector(function(d) { return d.date; }).left; // **

// Set the ranges
var x = d3.time.scale().range([0, width]);
var y = d3.scale.linear().range([height, 0]);

// Define the axes
var xAxis = d3.svg.axis().scale(x)
    .orient("bottom").ticks(5);

var yAxis = d3.svg.axis().scale(y)
    .orient("left").ticks(5);

// Define the line
var valueline = d3.svg.line()
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y(d.close); });

// Adds the svg canvas
var svg = d3.select("body")
    .append("svg")
        .attr("width", width + margin.left + margin.right)
        .attr("height", height + margin.top + margin.bottom)
    .append("g")
        .attr("transform",
              "translate(" + margin.left + "," + margin.top + ")");

var lineSvg = svg.append("g");                              // **********

var focus = svg.append("g")                                 // **********
    .style("display", "none");                              // **********
```

```
// Get the data
d3.csv("atad.csv", function(error, data) {                // **********
    data.forEach(function(d) {
        d.date = parseDate(d.date);
        d.close = +d.close;
    });

    // Scale the range of the data
    x.domain(d3.extent(data, function(d) { return d.date; }));
    y.domain([0, d3.max(data, function(d) { return d.close; })]);

    // Add the valueline path.
    lineSvg.append("path")                                // **********
        .attr("class", "line")
        .attr("d", valueline(data));

    // Add the X Axis
    svg.append("g")
        .attr("class", "x axis")
        .attr("transform", "translate(0," + height + ")")
        .call(xAxis);

    // Add the Y Axis
    svg.append("g")
        .attr("class", "y axis")
        .call(yAxis);

    // append the circle at the intersection                // **********
    focus.append("circle")                                // **********
        .attr("class", "y")                               // **********
        .style("fill", "none")                            // **********
        .style("stroke", "blue")                          // **********
        .attr("r", 4);                                    // **********

    // append the rectangle to capture mouse                // **********
    svg.append("rect")                                    // **********
        .attr("width", width)                             // **********
        .attr("height", height)                           // **********
        .style("fill", "none")                            // **********
        .style("pointer-events", "all")                   // **********
        .on("mouseover", function() { focus.style("display", null); })
        .on("mouseout", function() { focus.style("display", "none"); })
        .on("mousemove", mousemove);                      // **********

    function mousemove() {                                // **********
        var x0 = x.invert(d3.mouse(this)[0]),             // **********
```

```
            i = bisectDate(data, x0, 1),                    // **********
            d0 = data[i - 1],                               // **********
            d1 = data[i],                                   // **********
            d = x0 - d0.date > d1.date - x0 ? d1 : d0;      // **********

        focus.select("circle.y")                            // **********
            .attr("transform",                              // **********
                  "translate(" + x(d.date) + "," +          // **********
                                 y(d.close) + ")");         // **********
    }                                                       // **********

});

</script>
</body>
```

Description

You should be able to tell from the asterisks in the code above that there aren't too many changes and appart from a few at the start and middle, the majority are contained in a large block towards the end.

Starting with our first change

```
bisectDate = d3.bisector(function(d) { return d.date; }).left;
```

This is our function that will be called later in the code that returns a value in our array of data that corresponds to the horizontal position of the mouse pointer. Specifically it returns the date that falls to the left of the mouse cursor.

The `d3.bisector`[318] is an 'array method' that can use an *accessor* or *comparator* function to divide an array of objects. In this case our array of date values. In the code I have used the `d3.bisector` as an *accessor*, because I believe that it's simpler to do so for the point of explanation, but the downside is that I had to have my dates ordered in ascending order which is why I load a slightly different csv file later (`atad.csv`).

If your eyes glazed over slightly reading the previous paragraph, don't let that put you off. Like with so many things, just relax and let d3.js do the magic and remember that `d3.bisector` can find a value in an ordered array.

The next block of changes declares a couple of functions that we will use to add our elements to our graph;

[318] https://github.com/mbostock/d3/wiki/Arrays#wiki-d3_bisector

```
var lineSvg = svg.append("g");

var focus = svg.append("g")
    .style("display", "none");
```

We will use `lineSvg` to add our line for the line graph and `focus` will add our tooltip elements. it is possible to avoid using lineSvg, but this way of declaring the functions means that we can control which elements are on top of which on the screen. For instance, it would be a pretty sad affair if our tooltip was appearing under the line of the line graph (hard to read).

As we saw earlier, our data is being sourced from a different csv file (`atad.csv`).

```
d3.csv("atad.csv", function(error, data) {
```

This is because we need to have it in a compatible order (ascending) to allow our bisector function to operate correctly. So while the line may look the same as the simple graph version, the data is ordered in reverse (some may say that this is the way the original data should have been presented all along, but I suppose we can't always second guess the data we get).

We then make a small change to the script that appended the line to the graph and instead of using `svg.append`... we use our newly declared `lineSvg`.

```
    lineSvg.append("path")
        .attr("class", "line")
        .attr("d", valueline(data));
```

The final, larger block of code can be broken into 4 logical sections;

1. Adding the circle to the graph
2. Set the area that we use to capture our mouse movements
3. The clever maths that determines which date will be highlighted
4. Move the circle to the appropriate position

The last two points actually occur within a separate function, but for the purposes of explanation I'm happy that this is a logical division of labour for the script.

Adding the circle to the graph

Adding the circle to the graph is actually fairly simple;

```
focus.append("circle")
    .attr("class", "y")
    .style("fill", "none")
    .style("stroke", "blue")
    .attr("r", 4);
```

If you've followed any of the other examples in *D3 Tips and Tricks* there shouldn't be any surprises here (well, perhaps assigning a class to the circle (`y`) could count as mildly unusual).

Except for one small thing....

We don't place it anywhere on the graph! There is no x y coordinates and no translation of position. Nothing! Never fear. All we want to do at this stage is to create the element. In a few blocks of code time we will move the circle.

Set the area to capture the mouse movements

As we briefly covered earlier, the thing that makes this particular tooltip technique different is that we don't hover over an element to highlight the tooltip. Instead we move the mouse into an area which is *relevant* to the tooltip and it appears.

And its all thanks to the following code;

```
svg.append("rect")
    .attr("width", width)
    .attr("height", height)
    .style("fill", "none")
    .style("pointer-events", "all")
    .on("mouseover", function() { focus.style("display", null); })
    .on("mouseout", function() { focus.style("display", "none"); })
    .on("mousemove", mousemove);
```

Here we're adding a rectangle to the graph (`svg.append("rect")`) with the same height and width as our graph area (`.attr("width", width)` and `.attr("height", height)`) and we're making sure that there's no colour (fill) in it (`.style("fill", "none")`). Nothing too weird about all that.

Then we make sure that if any mouse events occur within the area that we capture them (`.style("pointer-events", "all")`). This is when things start to get interesting.

The first pointer event that we want to work with is `mouseover`;

```
    .on("mouseover", function() { focus.style("display", null); })
```

This line of code tells the script that when the mouse moves over the area of the rectangle of the area of the graph the display properties of the `focus` elements (remember that we appended our circle to `focus` earlier) are set to `null`. This might sound like a bit of a strange thing to do, since what we want to do is to make sure that when the mouse moves over the graph we want the `focus` elements to be displayed. but by setting the `display` style to `null` the default value for `display`

is enacted and this is `inline` which allows the elements to be rendered as normal. So why not use `inline` instead of `null`? Good question. I've tried it and it works without problem, but the original example that Mike Bostock used had the setting at `null` and I'll make the assumption that Mike knows something that I don't know about when to use `null` and when to use `inline` for a display style[319] (maybe some browser incompatibility issues?).

The reverse of making our `focus` element display display everything is being able to make it stop displaying everything. This is what happens in the next line;

```
.on("mouseout", function() { focus.style("display", "none"); })
```

Here, where the mouse moves off the area, the display properties for the `focus` element are turned off.

Lastly for this block, we need to capture the actions of the mouse as it moves on the graph area and move our tooltips as required. This is accomplished with the final line in the block...

```
.on("mousemove", mousemove);
```

... where if the mouse moves we call the `mousemove` function.

Determining which date will be highlighted

Once the `mousemove` function is called is carries out the last two steps in our code. The first of which is the clever maths that determines which point in our graph has the tooltip applied to it.

```
var x0 = x.invert(d3.mouse(this)[0]),
    i = bisectDate(data, x0, 1),
    d0 = data[i - 1],
    d1 = data[i],
    d = x0 - d0.date > d1.date - x0 ? d1 : d0;
```

The first line of this block is a dozy;

```
var x0 = x.invert(d3.mouse(this)[0]),
```

If we break it down the `d3.mouse(this)[0]` portion returns the x position on the screen of the mouse (`d3.mouse(this)[1]` would return the y position). Then the `x.invert` function is reversing the process that we use to map the domain (date) to range (position on screen). So it takes the position on the screen and converts it into an equivalent date!

For the adventurous amongst you, throw a `console.log(x0);` line into the `mousemove` function and check out the changing date/time as the cursor moves pixel by pixel (This will work for Google Chrome). Very cool.

Then we use our `bisectDate` function that we declared earlier to find the index of our `data` array that is close to the mouse cursor.

[319] http://www.w3schools.com/jsref/prop_style_display.asp

```
i = bisectDate(data, x0, 1),
```

It takes our data array and the date corresponding to the position of or mouse cursor and returns the index number of the data array which has a date that is higher than the cursor position.

Then we declare arrays that are subsets of our data array;

```
d0 = data[i - 1],
d1 = data[i],
```

`d0` is the combination of `date` and `close` that is in the data array at the index to the left of the cursor and `d1` is the combination of `date` and `close` that is in the data array at the index to the right of the cursor. In other words we now have two variables that know the value and date above and below the date that corresponds to the position of the cursor.

The final line in this segment declares a new array `d` that is represents the `date` and `close` combination that is closest to the cursor.

```
d = x0 - d0.date > d1.date - x0 ? d1 : d0;
```

It is using the magic JavaScript short hand for an `if` statement that is essentially saying if the distance between the mouse cursor and the `date` and `close` combination on the left is greater than the distance between the mouse cursor and the `date` and `close` combination on the right then `d` is an array of the `date` and `close` on the right of the cursor (`d1`). Otherwise `d` is an array of the `date` and `close` on the left of the cursor (`d0`).

This could be regarded as a fairly complicated little piece of code, but if you take the time to understand it, you will be surprised how elegant it appears. As we've seen before though, if you just want to believe that the d3.js magic is happening, that's fine.

Move the circle to the appropriate position

The final block of code that we'll check out takes the closest `date` / `close` combination that we've just worked out and moves the circle to that position;

```
focus.select("circle.y")
    .attr("transform",
          "translate(" + x(d.date) + "," +
                         y(d.close) + ")");
```

This is a pretty easy bit of code to follow. We select the circle (using the class `y` that we assigned to it earlier) and then move it using `translate` to the `date` / `close` position that we had just worked out was the closest.

Of course this is provision of the coordinates to the circle that we noticed was missing earlier in the code when we were appending it to the graph.

And there we have it. A simple circle positioned at the closest point to the mouse cursor when the cursor hovers over the graph.

Simple version of the favourite tooltip example

If we hadn't mentioned it earlier you might be thinking that this could possibly be the most complicated method for making most basic (read lame) tooltip ever. But you know there's more right? Right....? Read on.

Complex version

You've read to this point, so that's a sign that you're still interested. In that case, I recommend that you take a moment to check out the live example[320] of the graph that I'm going to describe.

More complicated favourite tooltip example

Here's a graph that when you move your mouse over it shows the closest intersection point on the graph with lines that extend the full width of the graph (great for comparing the level across the graph) and down to the x axis (to get a rough feel for the date). As well as this there is a subtle circle around the data point in question (as already explained in the previous section) and the actual date and value represented at the intersection point. As if that wasn't enough there is a nice little drop shadow effect under the text so that no matter what the background is you can read it. Nice.

The full code for this example is available online at bl.ocks.org[321] or GitHub[322]. It is also available as the files 'best-tooltip-coolio.html' and 'atad.csv' as a download with the book D3 Tips and Tricks (in a zip file) when you download the book from Leanpub[323].

[320] http://bl.ocks.org/d3noob/6eb506b129f585ce5c8a
[321] http://bl.ocks.org/d3noob/6eb506b129f585ce5c8a
[322] https://gist.github.com/d3noob/6eb506b129f585ce5c8a
[323] https://leanpub.com/D3-Tips-and-Tricks

Code / Explanation

Because the date at the tooltip needs to be formatted in a particular way we need to declare this appropriately;

```
formatDate = d3.time.format("%d-%b"),
```

Other than that everything is pretty normal until we get to the part where we start adding elements to our `focus` group (you remember we had the circle before? Now we're adding additional elements.).

```
// append the x line
focus.append("line")
    .attr("class", "x")
    .style("stroke", "blue")
    .style("stroke-dasharray", "3,3")
    .style("opacity", 0.5)
    .attr("y1", 0)
    .attr("y2", height);

// append the y line
focus.append("line")
    .attr("class", "y")
    .style("stroke", "blue")
    .style("stroke-dasharray", "3,3")
    .style("opacity", 0.5)
    .attr("x1", width)
    .attr("x2", width);

// append the circle at the intersection
focus.append("circle")
    .attr("class", "y")
    .style("fill", "none")
    .style("stroke", "blue")
    .attr("r", 4);

// place the value at the intersection
focus.append("text")
    .attr("class", "y1")
    .style("stroke", "white")
    .style("stroke-width", "3.5px")
    .style("opacity", 0.8)
    .attr("dx", 8)
    .attr("dy", "-.3em");
focus.append("text")
    .attr("class", "y2")
```

```
        .attr("dx", 8)
        .attr("dy", "-.3em");

    // place the date at the intersection
    focus.append("text")
        .attr("class", "y3")
        .style("stroke", "white")
        .style("stroke-width", "3.5px")
        .style("opacity", 0.8)
        .attr("dx", 8)
        .attr("dy", "1em");
    focus.append("text")
        .attr("class", "y4")
        .attr("dx", 8)
        .attr("dy", "1em");
```

Here you can see we're adding the x (horizontal) line and the y (vertical) line as well as the date and text values. Notice on the text values, there is a white drop shadow added first and then the text over the top. Another thing to note is that just like the position information, we don't actually put the text in here, this is simple a 'placeholder' for the element.

Then all we need to do is move all the new elements to the correct position and add the changing text where appropriate;

```
focus.select("circle.y")
    .attr("transform",
          "translate(" + x(d.date) + "," +
                         y(d.close) + ")");

focus.select("text.y1")
    .attr("transform",
          "translate(" + x(d.date) + "," +
                         y(d.close) + ")")
    .text(d.close);

focus.select("text.y2")
    .attr("transform",
          "translate(" + x(d.date) + "," +
                         y(d.close) + ")")
    .text(d.close);

focus.select("text.y3")
    .attr("transform",
          "translate(" + x(d.date) + "," +
                         y(d.close) + ")")
    .text(formatDate(d.date));
```

```
focus.select("text.y4")
    .attr("transform",
          "translate(" + x(d.date) + "," +
                         y(d.close) + ")")
    .text(formatDate(d.date));

focus.select(".x")
    .attr("transform",
          "translate(" + x(d.date) + "," +
                         y(d.close) + ")")
               .attr("y2", height - y(d.close));

focus.select(".y")
    .attr("transform",
          "translate(" + width * -1 + "," +
                         y(d.close) + ")")
               .attr("x2", width + width);
```

There's no big surprises here. Just an extension of what we accomplished with the circle earlier. The only part that looks semi-interesting is some of the application of the positioning of the x and y lines and this is more because of the points at which the lines start and finish.

Now this is unlikely to be the end solution for most people, but at least there are plenty of examples of different elements in there to play with and experiment on.

Enjoy!

Exploring Event Data by Combination Scatter Plot and Interactive Line Graphs

Purpose

In the process of implementing a method of measuring and displaying the passage of a cat through a cat-door (as described in the book 'Raspberry Pi: Measure, Record, Explore[324]') I built a graph that showed events indicated by both date and time on separate axes. It was then that I figured that this would be useful for exploring event data or data that exists as a series of date/time stamps that signify a particular 'thing as having occurred. In the cat door example it was the use of the door by the cat, but this is applicable to a huge range of data sets.

One that I thought of straight away was the dates and times that people downloaded this book. Leanpub has an API for accessing the history of book activity and I was able to download it and store it in a database for examination.

Ultimately what I developed was a scatter plot that shows the date of the events on the X axis and the time of the events on the Y axis. This was augmented by two line graphs that showed the accumulated sums of each axis on their respective sides.

Data Event Exploration

The full code for this example is available online at bl.ocks.org[325] or GitHub[326]. It is also available as the files 'book-downloads.html' and 'downloads.zip' (which contains downloads.json (it's zipped up because otherwise it's a bit too large for Leanpub)) as a download with the book D3 Tips and Tricks (in a zip file) when you download the book from Leanpub[327].

[324] https://leanpub.com/RPiMRE
[325] http://bl.ocks.org/d3noob/a0cbcddc6bf0eb9569fe
[326] https://gist.github.com/d3noob/a0cbcddc6bf0eb9569fe
[327] https://leanpub.com/D3-Tips-and-Tricks

To make the information slightly more accessible when the user hovers their mouse over the scatter plot there is an intersection of the position extrapolated to show the relationship to the other graphs and it presents the appropriate value of date, time and number downloaded by date and time.

This graph is a relatively complex combination of a range of different techniques presented in the book, including wrangling and nesting of data, combination of multiple graphs and the use of mouse movement to display tooltips and additional data.

The Code

The code is extremely lengthy, so in lieu of placing it in the book it can be found on bl.ocks.org[328] or Github[329]. It is liberally commented to assist readers and I will describe particular sections of the code below and hopefully that will help more where required.

Wrangling the data

The graph uses four sets of data.

1. The raw event data (an array called `events`)
2. The scatter plot data (an array called `data`)
3. The date graph data (an array called `dataDate`)
4. The time graph data (an array called `dataTime`)

The raw event data is ingested from an external JSON file using the standard `d3.json` call.

The data itself is simply a collection of dates.

```
{"dtg":"2013-01-24 09:10:59"},
{"dtg":"2013-01-24 09:17:37"},
{"dtg":"2013-01-24 09:48:48"},
{"dtg":"2013-01-24 15:01:59"},
{"dtg":"2013-01-24 18:11:44"},
{"dtg":"2013-01-24 18:47:05"},
{"dtg":"2013-01-24 18:47:23"},
{"dtg":"2013-01-24 19:55:53"},
{"dtg":"2013-01-24 22:37:39"},
{"dtg":"2013-01-25 01:22:48"},
{"dtg":"2013-01-25 06:37:38"},
{"dtg":"2013-01-25 08:28:20"},
```

Each date represents the time that a book was downloaded.

Once loaded we run a `forEach` over the file to put it in a format for manipulation into the remaining three data sets.

[328] http://bl.ocks.org/d3noob/a0cbcddc6bf0eb9569fe
[329] https://gist.github.com/d3noob/a0cbcddc6bf0eb9569fe

```
// parse and format all the event data
events.forEach(function(d) {
    d.dtg = d.dtg.slice(0,-4)+'0:00'; // get the 10 minute block
    dtgSplit = d.dtg.split(" ");      // split on the space
    d.date = dtgSplit[0];             // get the date seperatly
    d.time = dtgSplit[1];             // format the time
    d.number_downloaded = 1;          // Number of downloads
});
```

The first thing we do is to `slice` off the last four characters of the `dtg` string and replace them with `0:00`. This leave us with a set of `dtg` values that are only represented by the 10 minute window in which they were downloaded.

We then `split` the `dtg` string on the space that separates the date and the time and we designate one half `date` and the other half `time`.

Lastly we represent the number of books downloaded for each event as 1 (this helps us sum them up later).

Using the `events` data we create the data-set for the scatter plot (`data`) by nesting the information on the 10 minute `dtg` value of date/time and by summing the number of downloads;

```
var data = d3.nest()
    .key(function(d) { return d.dtg;})
    .rollup(function(d) {
        return d3.sum(d,function(g) {return g.number_downloaded; });
        })
    .entries(events);
```

We carry out a similar process for the date...

```
var dataDate = d3.nest()
    .key(function(d) { return d.date;})
    .rollup(function(d) {
        return d3.sum(d,function(g) {return g.number_downloaded; });
        })
    .entries(events);
```

... and the time;

```
var dataTime = d3.nest()
    .key(function(d) { return d.time;})
    .sortKeys(d3.ascending)
    .rollup(function(d) {
        return d3.sum(d,function(g) {return g.number_downloaded; });
        })
    .entries(events);
```

Sizing Everything Up

The size of the graph is determined by a number of fixed variables which are fairly self explanatory;

- `scatterplotHeight` (which is also the height of the time graph)
- `dateGraphHeight`
- `timeGraphWidth`

But we need to let the width of the scatter plot (and the date graph) be a function of the number of days that have been collected. This variable is handled by;

- `scatterplotWidth`

This set-up is handled in the following block of code;

```
var oneDay = 24*60*60*1000; // hours*minutes*seconds*milliseconds
var dateStart = d3.min(data, function(d) { return d.date; });
var dateFinish = d3.max(data, function(d) { return d.date; });
var numberDays = Math.round(Math.abs((dateStart.getTime() -
                          dateFinish.getTime())/(oneDay)));

var margin = {top: 20, right: 20, bottom: 20, left: 50},
    scatterplotHeight = 520,
    scatterplotWidth = numberDays * 1.5,
    dateGraphHeight = 220,
    timeGraphWidth = 220;
```

The overall size of the graphic (`height` and `width`) is therefore a combination of these variables;

```
var height = scatterplotHeight + dateGraphHeight,
    width = scatterplotWidth + timeGraphWidth;
```

The Scatter Plot

There is no real surprise with the scatter plot itself. The only thing slightly unusual is the use of a time scale for both the X and Y axes;

```
var x = d3.time.scale().range([0, scatterplotWidth]);
var y = d3.time.scale().range([0, scatterplotHeight]);
```

When the circles are drawn, the size of the circle is determined by the radius, which is the number of downloads multiplied by 1.5. I know that this is a bit of a visualization 'no-no' because the area of the circle should be representative of the number, not the radius, but I tried it both ways and to my simple way of viewing the data, the radius adjustment provided the best comparison.

```
svg.selectAll(".dot")
    .data(data)
  .enter().append("circle")
    .attr("class", "dot")
    .attr("r", function(d) { return d.number_downloaded*1.5; })
    .style("opacity", 0.3)
    .style("fill", "#e31a1c" )
    .attr("cx", function(d) { return x(d.date); })
    .attr("cy", function(d) { return y(d.time); });
```

I know that this is a topic of some academic debate, and it is fascinating, so here are both results for comparison;

Circle Area Representing Downloads

Circle Radius Representing Downloads

Date and Time Graphs

Both of these graphs are fairly routine. The time graph has the X and Y axes reversed from what would be ordinarily expected, but otherwise not much else to write home about.

Mouse Movement Information Display

This portion of the graph is an expansion of the 'Favourite tool tip' method from the previous section in this chapter. We expand the number of elements to update dynamically to about 10. All of which are designated with their own `class`.

We append the rectangle to capture the mouse movement over the scatterplot;

```
svg.append("rect")
    .attr("width", scatterplotWidth)
    .attr("height", scatterplotHeight)
    .style("fill", "none")
    .style("pointer-events", "all")
    .on("mouseover", function() { focus.style("display", null); })
    .on("mouseout", function() { focus.style("display", "none"); })
    .on("mousemove", mousemove);
```

We capture the position of the mouse and convert it to figures we can use to compare to our data;

```
function mousemove() {
    var xpos = d3.mouse(this)[0],
        x0 = x.invert(xpos),
        y0 = d3.mouse(this)[1],
        y1 = y.invert(y0),
        date1 = d3.mouse(this)[0];
```

And then we place our dynamic text and lines with our `focus.select` statements.

Labelling

The last order of business is to place some labels.

The location of labelling in this example is an interesting problem in itself. I'm personally torn between the desire to maintain simplicity and to ensure clarity. Hopefully what I have is enough to satisfy both requirements, but as always, each user and requirement will differ, so label as desired.

If there are additional parts of the code that you would like explained, please feel free to get in touch.

Difference Chart: Science vs Style.

Dear readers, please forgive me for including this example in *D3 Tips and Tricks*. While it demonstrates a really cool graphing technique, I have chosen to apply it to a topic that has a potential to raise a couple of sets of eyebrows in the form of Messrs Roger Peng and Jeff Leek. Both work at the Johns Hopkins Bloomberg School of Public Health where Roger is an Associate Professor of Biostatistics, and Jeff is an Associate Professor of Biostatistics and Oncology.

While both are doing amazing work to improve peoples health and well-being (amongst other things), both are also authors of highly successful books published by Leanpub. In particular Roger has written R Programming for Data Science[330] and Exploratory Data Analysis with R[331] while Jeff has penned The Elements of Data Analytic Style[332]. As we could anticipate, there is a possibility that there is something of a competitive element[333] to publishing for both gentlemen as they see the the number of downloads of their books climb ever higher.

While I would hate to promote an increase to these tensions, The opportunity was too attractive given that I had access to some data on the number of downloads that each of the books had been achieving and I really wanted to write about difference charts using d3.js (and the method of sourcing the data for the book Raspberry Pi: Measure, Record, Explore[334]).

So at the risk of providing some form of offence to these fine gentlemen or inciting an increased rivalry, I have forged ahead and hopefully the worst that will happen is that someone interested in d3.js will also find some interesting reading in *R Programming for Data Science*, *Exploratory Data Analysis with R* or *The Elements of Data Analytic Style*. Ultimately we should be left with a graph that will look something like this;

Science vs Style - Daily Leanpub Book Sales

[330] https://leanpub.com/rprogramming
[331] https://leanpub.com/exdata
[332] https://leanpub.com/datastyle
[333] https://twitter.com/d3noob/status/611227825685725184/photo/1
[334] https://leanpub.com/RPiMRE

Purpose

A difference chart is a variation on a bivariate area chart[335]. This is a line chart that includes two lines that are interlinked by filling the space between the lines. A difference chart as demonstrated in the example here[336] by Mike Bostock is able to highlight the differences between the lines by filling the area between them with different colours depending on which line is the greater value.

As Mike points out in his example, this technique harks back at least as far as William Playfair[337] when he was describing the time series of exports and imports of Denmark and Norway in 1786.

William Playfair's Time Series of Exports and Imports of Denmark and Norway

All that remains is for us to work out how d3.js can help us out by doing the job programmatically. The example that I use here is based on that of Mike Bostock's[338], with the addition of a few niceties in the form of a legend, a title, and some minor changes.

We will start with a simple example of the code and we will add blocks to finally arrive at the example with Legends and title.

[335] http://www.informit.com/articles/article.aspx?p=709139&seqNum=5
[336] http://bl.ocks.org/mbostock/3894205
[337] https://en.wikipedia.org/wiki/William_Playfair
[338] http://bl.ocks.org/mbostock/3894205

The Code

The following is the code for the simple difference chart. A live version is available online at bl.ocks.org[339] or GitHub[340]. It is also available as the files 'diff-basic.html' and 'downloads.csv' as a download with the book D3 Tips and Tricks (in a zip file) when you download the book from Leanpub[341].

```
<!DOCTYPE html>
<meta charset="utf-8">
<style>

body { font: 10px sans-serif;}

text.shadow {
  stroke: white;
  stroke-width: 2px;
  opacity: 0.9;
}

.axis path,
.axis line {
  fill: none;
  stroke: #000;
  shape-rendering: crispEdges;
}

.x.axis path { display: none; }

.area.above { fill: rgb(252,141,89); }
.area.below { fill: rgb(145,207,96); }

.line {
  fill: none;
  stroke: #000;
  stroke-width: 1.5px;
}

</style>
<body>
<script src="http://d3js.org/d3.v3.min.js"></script>
<script>

var title = "Science vs Style - Daily Leanpub Book Sales";
```

[339] http://bl.ocks.org/d3noob/8beea1d918ff4104f9ab
[340] https://gist.github.com/d3noob/8beea1d918ff4104f9ab
[341] https://leanpub.com/D3-Tips-and-Tricks

```
var margin = {top: 20, right: 20, bottom: 50, left: 50},
    width = 960 - margin.left - margin.right,
    height = 500 - margin.top - margin.bottom;

var parsedtg = d3.time.format("%Y-%m-%d").parse;

var x = d3.time.scale().range([0, width]);
var y = d3.scale.linear().range([height, 0]);

var xAxis = d3.svg.axis().scale(x).orient("bottom");
var yAxis = d3.svg.axis().scale(y).orient("left");

var lineScience = d3.svg.area()
    .interpolate("basis")
    .x(function(d) { return x(d.dtg); })
    .y(function(d) { return y(d["Science"]); });

var lineStyle = d3.svg.area()
    .interpolate("basis")
    .x(function(d) { return x(d.dtg); })
    .y(function(d) { return y(d["Style"]); });

var area = d3.svg.area()
    .interpolate("basis")
    .x(function(d) { return x(d.dtg); })
    .y1(function(d) { return y(d["Science"]); });

var svg = d3.select("body").append("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
  .append("g")
    .attr("transform",
        "translate(" + margin.left + "," + margin.top + ")");

d3.csv("downloads.csv", function(error, dataNest) {

  dataNest.forEach(function(d) {
      d.dtg = parsedtg(d.date_entered);
      d.downloaded = +d.downloaded;
  });

  var data = d3.nest()
      .key(function(d) {return d.dtg;})
      .entries(dataNest);

  data.forEach(function(d) {
```

```
      d.dtg = d.values[0]['dtg'];
      d["Science"] = d.values[0]['downloaded'];
      d["Style"] = d.values[1]['downloaded'];
  });

  for(i=data.length-1;i>0;i--) {
          data[i].Science   = data[i].Science  -data[(i-1)].Science ;
          data[i].Style     = data[i].Style    -data[(i-1)].Style ;
   }

  data.shift(); // Removes the first element in the array

    x.domain(d3.extent(data, function(d) { return d.dtg; }));
    y.domain([
//      d3.min(data, function(d) {
//          return Math.min(d["Science"], d["Style"]); }),
//      d3.max(data, function(d) {
//          return Math.max(d["Science"], d["Style"]); })
      0,1400
    ]);

    svg.datum(data);

    svg.append("clipPath")
        .attr("id", "clip-above")
      .append("path")
        .attr("d", area.y0(0));

    svg.append("clipPath")
        .attr("id", "clip-below")
      .append("path")
        .attr("d", area.y0(height));

    svg.append("path")
        .attr("class", "area above")
        .attr("clip-path", "url(#clip-above)")
        .attr("d", area.y0(function(d) { return y(d["Style"]); }));

    svg.append("path")
        .attr("class", "area below")
        .attr("clip-path", "url(#clip-below)")
        .attr("d", area.y0(function(d) { return y(d["Style"]); }));

    svg.append("path")
        .attr("class", "line")
        .style("stroke", "darkgreen")
```

```
        .attr("d", lineScience);

    svg.append("path")
        .attr("class", "line")
        .style("stroke", "red")
        .attr("d", lineStyle);

    svg.append("g")
        .attr("class", "x axis")
        .attr("transform", "translate(0," + height + ")")
        .call(xAxis);

    svg.append("g")
        .attr("class", "y axis")
        .call(yAxis);

});

</script>
</body>
```

A sample of the associated csv file (downloads.csv) is formatted as follows;

```
date_entered,downloaded,book_name
2015-04-19,5481,R Programming for Data Science
2015-04-19,23751,The Elements of Data Analytic Style
2015-04-20,5691,R Programming for Data Science
2015-04-20,23782,The Elements of Data Analytic Style
2015-04-21,6379,R Programming for Data Science
2015-04-21,23820,The Elements of Data Analytic Style
2015-04-22,7281,R Programming for Data Science
2015-04-22,23857,The Elements of Data Analytic Style
2015-04-23,7554,R Programming for Data Science
2015-04-23,23881,The Elements of Data Analytic Style
2015-04-24,9331,R Programming for Data Science
2015-04-24,23932,The Elements of Data Analytic Style
```

Description

The graph has some portions that are common to the simple line graph example.

We start the HTML file, load some styling for the upcoming elements, set up the margins, time formatting scales, ranges and axes.

Because the graph is composed of two lines we need to declare two separate line functions;

```
var lineScience = d3.svg.area()
    .interpolate("basis")
    .x(function(d) { return x(d.dtg); })
    .y(function(d) { return y(d["Science"]); });

var lineStyle = d3.svg.area()
    .interpolate("basis")
    .x(function(d) { return x(d.dtg); })
    .y(function(d) { return y(d["Style"]); });
```

To fill an area we declare an area function using one of the lines as the baseline (`y1`) and when it comes time to fill the area later in the script we declare `y0` separately to define the area to be filled as an intersection of two paths.

```
var area = d3.svg.area()
    .interpolate("basis")
    .x(function(d) { return x(d.dtg); })
    .y1(function(d) { return y(d["Science"]); });
```

In this instance we are using the green 'Science' line as the `y1` line.

The svg area is then set up using the height, width and margin values and we load our csv files with our number of downloads for each book. We then carry out a standard `forEach` to ensure that the time and numerical values are formatted correctly.

Nesting the data

The data that we are starting with is formatted in a way that we could reasonably expect data to be available in this instance where a value is saved for distinct elements on an element by element basis. This style of recording data makes it easy to add new elements into the data stream or a database rather than relying on having them as discrete columns.

```
date_entered,downloaded,book_name
2015-04-19,5481,R Programming for Data Science
2015-04-19,23751,The Elements of Data Analytic Style
2015-04-20,5691,R Programming for Data Science
2015-04-20,23782,The Elements of Data Analytic Style
2015-04-21,6379,R Programming for Data Science
2015-04-21,23820,The Elements of Data Analytic Style
```

In this case, we will need to 'pivot' the data to produce a multi-column representation where we have a single row for each date, and the number of downloads for each book as separate columns as follows;

```
date_entered,R Programming for Data Science,The Elements of Data Analytic Style
2015-04-19,5481,23751
2015-04-20,5691,23782
2015-04-21,6379,23820
```

This can be achieved using the d3 nest function.

```
var data = d3.nest()
    .key(function(d) {return d.dtg;})
    .entries(dataNest);
```

We declare our new array's name as `data` and we initiate the `nest` function;

```
var data = d3.nest()
```

We assign the `key` for our new array as `dtg`. A 'key' is like a way of saying "*This is the thing we will be grouping on*". In other words our resultant array will have a single entry for each unique date (`dtg`) which will have the values of the number of downloaded books associated with it.

```
.key(function(d) {return d.dtg;})
```

Then we tell the nest function which data array we will be using for our source of data.

```
}).entries(dataNest);
```

Wrangle the data

Once we have our pivoted data we can format it in a way that will suit the code for the visualisation. This involves storing the values for the 'Science' and 'Style' variables as part of a named index.

```
data.forEach(function(d) {
    d.dtg = d.values[0]['dtg'];
    d["Science"] = d.values[0]['downloaded'];
    d["Style"] = d.values[1]['downloaded'];
});
```

We then loop through the 'Science' and 'Style' array to convert the incrementing value of the total number of downloads into a value of the number that have been downloaded each day;

```
for(i=data.length-1;i>0;i--) {
        data[i].Science   = data[i].Science  -data[(i-1)].Science ;
        data[i].Style     = data[i].Style    -data[(i-1)].Style ;
 }
```

Finally because we are adjusting from total downloaded to daily values we are left with an orphan value that we need to remove from the front of the array;

```
data.shift();
```

Cheating with the domain

The observant d3.js reader will have noticed that the setting of the `y` domain has a large section commented out;

```
    x.domain(d3.extent(data, function(d) { return d.dtg; }));
    y.domain([
//      d3.min(data, function(d) {
//          return Math.min(d["Science"], d["Style"]); }),
//      d3.max(data, function(d) {
//          return Math.max(d["Science"], d["Style"]); })
      0,1400
    ]);
```

That's because I want to be able to provide an ideal way for the graph to represent the data in an appropriate range, but because we are using the `basis` smoothing modifier, and the data is 'peaky', there is a tendency for the y scale to be fairy broad and the resultant graph looks a little lost;

Using automatic range

Alternatively, we could remove the smoothing and let the true data be shown;

Using automatic range and removing the basis smoothing

It should be argued that this is a truer representation of the data, but in this case I feel comfortable sacrificing accuracy for aesthetics (what have I become?).

Therefore, the domain for the `y` axis is set manually to between 0 and 1400, but feel free to remove that at the point when you introduce your own data :-).

`data` VS `datum`

One small line gets its own section. That line is;

```
svg.datum(data);
```

A casual d3.js user could be forgiven for thinking that this doesn't seem too fearsome a line, but it has hidden depths.

As Mike Bostock explains here[342], if we want to bind data to elements as a group we would be `*.data`, but if we want to bind that data to individual elements, we should use `*.datum`.

It's a function of how the data is stored. If there is an expectation that the data will be dynamic then `data` is the way to go since it has the feature of preparing enter and exit selections. If the data is static (it won't be changing) then `datum` is the way to go.

In our case we are assigning data to individual elements and as a result we will be using `datum`.

Setting up the `clipPath`s

The `clipPath`[343] operator is used to define an area that is used to create a shape by intersecting one area with another.

[342] http://bost.ocks.org/mike/selection/#data
[343] https://developer.mozilla.org/en-US/docs/Web/SVG/Element/clipPath

In our case we are going to set up two clip paths. One is the area above the green 'Science' line (which we defined earlier as being the `y1` component of an area selection);

the 'clip-above' clip path

This is declared via this portion of the code;

```
svg.append("clipPath")
    .attr("id", "clip-above")
  .append("path")
    .attr("d", area.y0(0));
```

Then we set up the clip path that will exist for the area below the green 'Science' line ;

The 'clip-below' clip path

This is declared via this portion of the code;

```
svg.append("clipPath")
    .attr("id", "clip-below")
  .append("path")
    .attr("d", area.y0(height));
```

Each of these paths has an 'id' which can be subsequently used by the following code.

Clipping and adding the areas

Now we come to clipping our shape and filling it with the appropriate colour.

We do this by having a shape that represents the area between the two lines and applying our clip path for the values above and below our reference line (the green 'Science' line). Where the two intersect, we fill it with the appropriate colour. The code to fill the area above the reference line is as follows;

```
svg.append("path")
    .attr("class", "area above")
    .attr("clip-path", "url(#clip-above)")
    .attr("d", area.y0(function(d) { return y(d["Style"]); }));
```

Here we have two lines that are defining the shape between the two science and style lines;

```
svg.append("path")
    ....
    ....
    .attr("d", area.y0(function(d) { return y(d["Style"]); }));
```

If we were to look at the shape that this produces it would look as follows (greyed out for highlighting);

The shape between the science and style lines

We apply a class to the shape so that is filled with the colour that we want;

```
.attr("class", "area above")
```

.. and apply the clip path so that only the areas that intersect the two shapes are filled with the appropriate colour;

```
.attr("clip-path", "url(#clip-above)")
```

Here the intersection of those two shapes is shown as pink;

The intersection of the shapes

Then we do the same for the area below;

```
svg.append("path")
    .attr("class", "area below")
    .attr("clip-path", "url(#clip-below)")
    .attr("d", area.y0(function(d) { return y(d["Style"]); }));
```

With the corresponding areas showing the intersection of the two shapes coloured differently;

The intersection of the shapes

Draw the lines and the axes

The final part of our basic difference chart is to draw in the lines over the top so that they are highlighted and to add in the axes;

```
svg.append("path")
    .attr("class", "line")
    .style("stroke", "darkgreen")
    .attr("d", lineScience);

svg.append("path")
    .attr("class", "line")
    .style("stroke", "red")
    .attr("d", lineStyle);

svg.append("g")
    .attr("class", "x axis")
    .attr("transform", "translate(0," + height + ")")
    .call(xAxis);

svg.append("g")
```

```
        .attr("class", "y axis")
        .call(yAxis);
```

Et viola! we have our difference chart!

The basic difference chart

As mentioned earlier, the code for the simple difference chart is available online at bl.ocks.org[344] or GitHub[345]. It is also available as the files 'diff-basic.html' and 'downloads.csv' as a download with the book D3 Tips and Tricks (in a zip file) when you download the book from Leanpub[346].

Adding a bit more to our difference chart.

The chart itself is a thing of beauty, but given the subject matter (it's describing two books after all) we should include a bit more information on what it is we're looking at and provide some links so that a fascinated viewer of the graphs can read the books!

Add a Y axis label

Because it's not immediately obvious what we're looking at on the Y axis we should add in a nice subtle label on the Y axis;

[344] http://bl.ocks.org/d3noob/8beea1d918ff4104f9ab
[345] https://gist.github.com/d3noob/8beea1d918ff4104f9ab
[346] https://leanpub.com/D3-Tips-and-Tricks

```
svg.append("g")
    .attr("class", "y axis")
    .call(yAxis)
  .append("text")
    .attr("transform", "rotate(-90)")
    .attr("y", 6)
    .attr("dy", ".71em")
    .style("text-anchor", "end")
    .text("Daily Downloads from Leanpub");
```

Add a title

Every graph should have a title. The following code adds this to the top(ish) centre of the chart and provides a white drop-shadow for readability;

```
// ******* Title Block ********
svg.append("text") // Title shadow
  .attr("x", (width / 2))
  .attr("y", 50 )
  .attr("text-anchor", "middle")
  .style("font-size", "30px")
  .attr("class", "shadow")
  .text(title);

svg.append("text") // Title
  .attr("x", (width / 2))
  .attr("y", 50 )
  .attr("text-anchor", "middle")
  .style("font-size", "30px")
  .style("stroke", "none")
  .text(title);
```

Adding the legend

A respectable legend in this case should provide visual context of what it is describing in relation to the graph (by way of colour) and should actually name the book. We can also go a little bit further and provide a link to the books in the legend so that potential readers can access them easily.

Firstly the rectangles filled with the right colour, sized appropriately and arranged just right;

```
var block = 300;   // rectangle width and position

svg.append("rect") // Style Legend Rectangle
  .attr("x", ((width / 2)/2)-(block/2))
  .attr("y", height+(margin.bottom/2) )
  .attr("width", block)
  .attr("height", "25")
  .attr("class", "area above");

svg.append("rect") // Science Legend Rectangle
  .attr("x", ((width / 2)/2)+(width / 2)-(block/2))
  .attr("y", height+(margin.bottom/2) )
  .attr("width", block)
  .attr("height", "25")
  .attr("class", "area below");
```

Then we add the text (with a drop-shadow) and a link;

```
svg.append("text") // Style Legend Text shadow
  .attr("x", ((width / 2)/2))
  .attr("y", height+(margin.bottom/2) + 5)
  .attr("dy", ".71em")
  .attr("text-anchor", "middle")
  .style("font-size", "18px")
  .attr("class", "shadow")
  .text("The Elements of Data Analytic Style");

svg.append("text") // Science Legend Text shadow
  .attr("x", ((width / 2)/2)+(width / 2))
  .attr("y", height+(margin.bottom/2) + 5)
  .attr("dy", ".71em")
  .attr("text-anchor", "middle")
  .style("font-size", "18px")
  .attr("class", "shadow")
  .text("R Programming for Data Science");

svg.append("a")
  .attr("xlink:href", "https://leanpub.com/datastyle")
  .append("text") // Style Legend Text
  .attr("x", ((width / 2)/2))
  .attr("y", height+(margin.bottom/2) + 5)
  .attr("dy", ".71em")
  .attr("text-anchor", "middle")
  .style("font-size", "18px")
  .style("stroke", "none")
  .text("The Elements of Data Analytic Style");
```

```
svg.append("a")
  .attr("xlink:href", "https://leanpub.com/rprogramming")
  .append("text") // Science Legend Text
  .attr("x", ((width / 2)/2)+(width / 2))
  .attr("y", height+(margin.bottom/2) + 5)
  .attr("dy", ".71em")
  .attr("text-anchor", "middle")
  .style("font-size", "18px")
  .style("stroke", "none")
  .text("R Programming for Data Science");
```

I'll be the first to admit that this could be done more efficiently with some styling via css, but then it would leave nothing for the reader to try :-).

Link the areas

As a last touch we can include the links to the respective books in the shading for the graph itself;

```
svg.append("a")
  .attr("xlink:href", "https://leanpub.com/datastyle")
  .append("path")
    .attr("class", "area above")
    .attr("clip-path", "url(#clip-above)")
    .attr("d", area.y0(function(d) { return y(d["Style"]); }));

svg.append("a")
  .attr("xlink:href", "https://leanpub.com/rprogramming")
  .append("path")
    .attr("class", "area below")
    .attr("clip-path", "url(#clip-below)")
    .attr("d", area.y0(function(d) { return y(d["Style"]); }));
```

Perhaps not strictly required, but a nice touch none the less.

The final result

And here it is;

The full difference chart

The code for the full difference chart is available online at bl.ocks.org[347] or GitHub[348]. It is also available as the files 'diff-full.html' and 'downloads.csv' as a download with the book D3 Tips and Tricks (in a zip file) when you download the book from Leanpub[349].

[347] http://bl.ocks.org/d3noob/f6d8588a0aa3a7503f57
[348] https://gist.github.com/d3noob/f6d8588a0aa3a7503f57
[349] https://leanpub.com/D3-Tips-and-Tricks

Crossfilter, dc.js and d3.js for Data Discovery

The ability to interact with visual data is the third step on the road to data nirvana in my humble opinion.

- Step 1: Raw data
- Step 2: Visualize data
- Step 3: Interact with data

But I think that there might be a 4th step where data is a more fluid construct. Where the influences of interaction have a more profound impact on how information is presented and perceived. I think that the visualization tools that we're going to explore in this chapter take that 4th step.

- Step 4: Data immersion

The tools we're going to use are not the only way that we can achieve the effect of immersion, but they are simple enough for me to use and they incorporate d3.js at their core.

Introduction to Crossfilter

Crossfilter is a JavaScript library for exploring large datasets that include many variables in the browser. It supports extremely fast interactions with concurrent views and was built to power analytics for Square Register[350] so that online merchants can slice and dice their payment history fluidly. It was developed for Square[351] by (amongst other people) the ever tireless Mike Bostock and was released under the Apache Licence[352].

Crossfilter provides a map-reduce function to data using 'dimensions' and 'groups'. Map-reduce is an interesting concept itself and it's useful to understand it in a basic form to understand crossfilter better.

Map-reduce

Wikipedia tells us[353] that "*MapReduce is a programming model for processing large data sets with a parallel, distributed algorithm on a cluster*". Loosely translated into language I can understand, I think of a large data set having one dimension 'mapped' or loaded into memory ready to be

[350] https://squareup.com/register
[351] https://squareup.com/
[352] http://www.apache.org/licenses/LICENSE-2.0.html
[353] https://en.wikipedia.org/wiki/MapReduce

worked on. In practical terms, this could be an individual column of data from a larger group of information. This column of data has 'key' values which we can define as being distinct. In the case of the data below, this could be earthquake magnitudes.

FID	publicid	origintime	depth	magnitude	longitude	latitude
quake.2013p550753	2013p550753	2013-07-23T18:41:11.707	7.9883	2.2	174.4298	-41.5313
quake.2013p550747	2013p550747	2013-07-23T18:38:02.481	11.6797	1.8	174.414	-41.5181
quake.2013p550725	2013p550725	2013-07-23T18:26:30.229	8.75	3.5	175.5516	-40.0264
quake.2013p550724	2013p550724	2013-07-23T18:25:52.618	11.2109	4.3	174.2292	-41.6837
quake.2013p550719	2013p550719	2013-07-23T18:23:04.562	5.1172	1.9	174.0623	-41.6946
quake.2013p550707	2013p550707	2013-07-23T18:16:41.795	18.0078	2.7	174.1772	-41.7113
quake.2013p550696	2013p550696	2013-07-23T18:11:13.808	7.6953	2.3	174.3964	-41.6919
quake.2013p550685	2013p550685	2013-07-23T18:05:20.677	140.4688	2.1	175.4547	-38.8079
quake.2013p550676	2013p550676	2013-07-23T18:01:08.435	21.2891	3.4	174.2831	-41.6327
quake.2013p550656	2013p550656	2013-07-23T17:50:01.564	10.0977	2.5	174.3607	-41.6244
quake.2013p550639	2013p550639	2013-07-23T17:40:58.613	106.4844	3.3	167.5654	-45.1528
quake.2013p550626	2013p550626	2013-07-23T17:33:41.661	5.0586	2.4	174.2668	-41.675
quake.2013p550623	2013p550623	2013-07-23T17:32:15.04	5.0586	2.3	174.3556	-41.5858
quake.2013p550609	2013p550609	2013-07-23T17:24:30.87	5.1172	2.2	174.2196	-41.6342
quake.2013p550598	2013p550598	2013-07-23T17:18:53.637	5.1172	2.0	174.3878	-41.529
quake.2013p550587	2013p550587	2013-07-23T17:13:47.503	5.6445	3.5	174.2576	-41.6488
quake.2013p550576	2013p550576	2013-07-23T17:06:55.593	19.8828	2.4	174.1842	-41.5197
quake.2013p550558	2013p550558	2013-07-23T16:57:41.898	5.3516	2.7	174.2321	-41.6229
quake.2013p550549	2013p550549	2013-07-23T16:52:50.013	5.0586	2.4	174.2858	-41.6297
quake.2013p550545	2013p550545	2013-07-23T16:50:42.2	25.7422	2.2	176.732	-39.5607
quake.2013p550538	2013p550538	2013-07-23T16:46:44.168	14.1406	2.1	174.3338	-41.5187
quake.2013p550529	2013p550529	2013-07-23T16:42:19.669	18.9453	2.5	174.1726	-41.7112
quake.2013p550514	2013p550514	2013-07-23T16:34:19.725	5.0586	2.3	174.2362	-41.6261
quake.2013p550510	2013p550510	2013-07-23T16:32:12.908	154.7129	3.8	178.667	-35.874
quake.2013p550507	2013p550507	2013-07-23T16:31:20.374	11.8555	3.6	174.3866	-41.5317

Mapping a Single Dimension of Data

The reduce function then takes that dimension and 'reduces' it by grouping it according to a specific aspect. For instance in the example above we may want to group each unique value of magnitude (by counting how many occurrences of each there are) to know how many earthquakes of a specific magnitude have taken place. Leaving us with a very specific subset of our data.

```
Magnitude Count
2.6       63
2.7       134
2.8       292
2.9       299
3.0       378
3.1       351
3.2       403
3.3       455
3.4       512
3.5       688
```

Please don't think that this is the sum total of information you need to know to be the master of map-reduce. This is a ridiculously simplistic view which is only intended to supply enough information to get you familiar with the way that we will use crossfilter later :-).

What can crossfilter do?

The best way to get a feel for the capabilities of crossfilter is to visit the demo page for crossfilter[354] and to play with their example.

Crossfilter Demo Page

Here we are presented with five separate views of a data set that represents flight records demonstrating airline on-time performance. There are 231,083 flight records in the database being used, so getting that rendered in a web page is no small feat in itself.

The bottom view is a table showing data for individual flights. The top, left view is of the number of flights that occur at a specific hour of the day.

Flights at a Specific Hour of the Day

The top, middle graph shows the amount of delay for flights grouped in 10 minute intervals.

[354] http://square.github.io/crossfilter/

Flights Delay in 10 Minute Intervals

The top, right graph shows the distance covered by each flight grouped in 50 mile chunks.

Flights Delay in 10 Minute Intervals

The wider bar graph in the second row shows the number of flights per day.

Flights per Day

This particular graph is the first to give a hint at how cool this visualization really is, because it includes a section in the middle of the graph which is selected with 'handles' on either side of the selection. You can move these handles with a mouse and as a result you will find all the data represented in the other graphs adjusting dynamically to follow your selection.

This same feature is available in all the graphs. So you are able to filter dynamically and have the results presented virtually instantaneously. This is where you can start to have fun and discover things that might not be immediately obvious.

For instance, if we select only the flights that arrived late, we can see a marked skew in the time of day. Does this mean that flights that are delayed will typically be in the late evening?

Arrival Delay and Time of Day

So this is why tools like crossfilter are cool. All we need to do now is learn how to make them ourselves :-).

Introduction to dc.js

Why, if we've just explored the benefits of crossfilter are we now introducing a completely different JavaScript library (dc.js)?

Well, crossfilter isn't a library that's designed to draw graphs. It's designed to manipulate data. D3.js is a library that's designed to manipulate graphical objects (and more) on a web page. The two of them will work really well together, but the barrier to getting data onto a web page can be slightly daunting because the combination of two non-trivial technologies can be difficult to achieve.

This is where dc.js[355] comes in. It was developed by Nick Qi Zhu[356] and the first version was released on the 7th of July 2012.

Dc.js is designed to be an enabler for both libraries. Taking the power of crossfilter's data manipulation capabilities and integrating the graphical capabilities of d3.js.

It is designed to provide access to a range of different chart types in a relatively easy to use fashion. It is more limited in the range of options available for graphical design in this respect than d3.js, but the simplicity that it provides for creating pages using crossfiltered data is a real benefit if you're anything like me and need all the help you can get.

The different (generic) types of chart that dc.js supports are

- Bar Chart
- Pie Chart
- Row Chart
- Line Chart
- Bubble Chart
- Geo Choropleth Chart

[355] http://nickqizhu.github.io/dc.js/
[356] https://github.com/NickQiZhu

- Data Table

All these examples come with a range of options which we will cover in greater depth in later sections.

My initial sources of information for developing the examples here came primarily from;

- Nick Zhu's examples[357]
- Rusty Klophaus' blog post on crossfilter[358]
- Eamonn O'Loughlin's blog post on dc.js[359]

Bar Chart

This is a standard bar chart.

Bar Chart Example

Pie Chart

This is a standard pie chart. The examples below are from one of Nick Zhu's dc.js example pages[360].

Pie Chart Examples

[357] http://nickqizhu.github.io/dc.js/
[358] http://blog.rusty.io/2012/09/17/crossfilter-tutorial/
[359] https://becomingadatascientist.wordpress.com/tag/crossfilter-js/
[360] http://nickqizhu.github.io/dc.js/

Row Chart

The row chart is a horizontal version of a bar chart, but with the ability to represent discrete values and to select them for filtering by clicking on them.

Row Chart Example

Line Chart

Standard line chart.

Line Chart Example

Bubble Chart

The bubble chart is a derivative of a scatter plot with control over x axis position, y axis position, bubble radius and colour.

Bubble Chart Example

Geo Choropleth Chart

A Choropleth map is one where areas are shaded or patterned in proportion to the measurement of a variable being displayed on the map, such as population density or per-capita income. The example below is from one of Nick Zhu's dc.js example pages[361]

Geo Choropleth Chart Example

Data Table

A data table is a simple table made up of data elements derived from the information loaded.

[361] http://nickqizhu.github.io/dc.js/vc/

DTG	Lat	Long	Depth	Magnitude	Google Map	OSM Map
List of all earthquakes corresponding to the filters						
Tue Jul 23 2013 17:50:01 GMT+1200 (New Zealand Standard Time)	-41.6244	174.3607	10	2.5	Google Map	OSM Map
Tue Jul 23 2013 18:01:08 GMT+1200 (New Zealand Standard Time)	-41.6327	174.2831	21	3.4	Google Map	OSM Map
Tue Jul 23 2013 18:05:20 GMT+1200 (New Zealand Standard Time)	-38.8079	175.4547	140	2.1	Google Map	OSM Map
Tue Jul 23 2013 18:11:13 GMT+1200 (New Zealand Standard Time)	-41.6919	174.3964	8	2.3	Google Map	OSM Map
Tue Jul 23 2013 18:16:41 GMT+1200 (New Zealand Standard Time)	-41.7113	174.1772	18	2.7	Google Map	OSM Map
Tue Jul 23 2013 18:23:04 GMT+1200 (New Zealand Standard Time)	-41.6946	174.0623	5	1.9	Google Map	OSM Map

Data Table Example

Bare bones structure for dc.js and crossfilter page

To learn some of the capabilities of dc.js and crossfilter we will start with a rudimentary template and build chart examples as we go.

The template we'll start with will load d3.js, crossfilter.js, dc.js, jquery.js and bootstrap.js. We will be including bootstrap as it provides lots of nice capabilities for fine tuning layout and styling as laid out in the chapter on using bootstrap. Since bootstrap depends on jquery, we have to load that as well.

We'll also load cascading style sheets for bootstrap and dc.js.

The template will load a csv file with earthquake data sourced from New Zealand's Geonet[362] site over a date range that covers a period of reasonable activity in July 2013.

In its bare bones form we will present only a data table with some values from the csv file. When we begin to add charts, we will see this table adjust dynamically.

We'll move through the explanation of the code in a similar process to the other examples in the book. Where there are areas that we have covered before, I will gloss over some details on the understanding that you will have already seen them explained in other sections.

The full code for this example can be found on github[363] or in the code samples bundled with this book (dcjs-examples.html, dc.js, dc.css, crossfilter.js, jquery-1.9.1.min.js, bootstrap.min.js, bootstrap.min.css and quakes.csv). A live example can be found on bl.ocks.org[364].

```
<!DOCTYPE html>
<html lang='en'>
<head>
  <meta charset='utf-8'>

  <title>dc.js Experiment</title>

  <script src="http://d3js.org/d3.v3.min.js"></script>
  <script src='crossfilter.js' type='text/javascript'></script>
  <script src='dc.js' type='text/javascript'></script>
  <script src='jquery-1.9.1.min.js' type='text/javascript'></script>
  <script src='bootstrap.min.js' type='text/javascript'></script>

  <link href='bootstrap.min.css' rel='stylesheet' type='text/css'>
  <link href='dc.css' rel='stylesheet' type='text/css'>

  <style type="text/css"></style>
</head>

<body>
```

[362] http://geonet.org.nz/
[363] https://gist.github.com/d3noob/6584483
[364] http://bl.ocks.org/d3noob/6584483

```
<div class='container' style='font: 12px sans-serif;'>
  <div class='row'>
        <div class='span12'>
      <table class='table table-hover' id='dc-table-graph'>
        <thead>
          <tr class='header'>
            <th>DTG</th>
            <th>Lat</th>
            <th>Long</th>
            <th>Depth</th>
            <th>Magnitude</th>
            <th>Google Map</th>
            <th>OSM Map</th>
          </tr>
        </thead>
      </table>
        </div>
  </div>
</div>

<script>

// Create the dc.js chart objects & link to div
var dataTable = dc.dataTable("#dc-table-graph");

// load data from a csv file
d3.csv("quakes.csv", function (data) {

  // format our data
  var dtgFormat = d3.time.format("%Y-%m-%dT%H:%M:%S");

  data.forEach(function(d) {
    d.dtg   = dtgFormat.parse(d.origintime.substr(0,19));
    d.lat   = +d.latitude;
    d.long  = +d.longitude;
    d.mag   = d3.round(+d.magnitude,1);
    d.depth = d3.round(+d.depth,0);
  });

  // Run the data through crossfilter and load our 'facts'
  var facts = crossfilter(data);

  // Create dataTable dimension
  var timeDimension = facts.dimension(function (d) {
    return d.dtg;
```

```
  });

  // Setup the charts

  // Table of earthquake data
  dataTable.width(960).height(800)
    .dimension(timeDimension)
        .group(function(d) { return "Earthquake Table"
         })
        .size(10)
    .columns([
      function(d) { return d.dtg; },
      function(d) { return d.lat; },
      function(d) { return d.long; },
      function(d) { return d.depth; },
      function(d) { return d.mag; },
      function(d) {
          return '<a href=\"http://maps.google.com/maps?z=12&t=m&q=loc:' +
              d.lat + '+' + d.long + "\" target=\"_blank\">Google Map</a>"},
      function(d) {
          return '<a href=\"http://www.openstreetmap.org/?mlat=' +
              d.lat + '&mlon=' + d.long +'&zoom=12'+
              "\" target=\"_blank\"> OSM Map</a>"}
    ])
    .sortBy(function(d){ return d.dtg; })
    .order(d3.ascending);

  // Render the Charts
  dc.renderAll();

});

</script>

</body>
</html>
```

The first part of the code starts the html file and inside the `<head>` segment loads our JavaScript and css files

```
<!DOCTYPE html>
<html lang='en'>
<head>
  <meta charset='utf-8'>

  <title>dc.js Experiment</title>

  <script src="http://d3js.org/d3.v3.min.js"></script>
  <script src='crossfilter.js' type='text/javascript'></script>
  <script src='dc.js' type='text/javascript'></script>
  <script src='jquery-1.9.1.min.js' type='text/javascript'></script>
  <script src='bootstrap.min.js' type='text/javascript'></script>

  <link href='bootstrap.min.css' rel='stylesheet' type='text/css'>
  <link href='dc.css' rel='stylesheet' type='text/css'>

  <style type="text/css"></style>
</head>
```

It's worth noting that the order of loading the files is important. The `jquery-1.9.1.min.js` file must be loaded before the `bootstrap.min.js` file or it just won't work.

From here we move into the section where we set up our page to load our bootstrap grid layout for the table.

```
<div class='container' style='font: 12px sans-serif;'>
  <div class='row'>
    <div class='span12'>
      <table class='table table-hover' id='dc-table-graph'>
        <thead>
          <tr class='header'>
            <th>DTG</th>
            <th>Lat</th>
            <th>Long</th>
            <th>Depth</th>
            <th>Magnitude</th>
            <th>Google Map</th>
            <th>OSM Map</th>
          </tr>
        </thead>
      </table>
    </div>
  </div>
</div>
```

It might look a little complicated, but if you have a look through the bootstrap chapter (where we cover using the bootstrap grid layout), you will find it no problem at all.

The important features to note are that we have declared an ID selector for our table `id='dc-table-graph'` and we have set a series of headers for the table; DTG, Lat, Long, Depth, Magnitude, Google Map and OSM Map.

We have also included some bootstrap styling for the table by including the `class='table table-hover'` portion of the code. With that styling included our table looks like this;

DTG	Lat	Long	Depth	Magnitude	Google Map	OSM Map
Earthquake Table						
Tue Jul 23 2013 17:50:01 GMT+1200 (New Zealand Standard Time)	-41.6244	174.3607	10	2.5	Google Map	OSM Map
Tue Jul 23 2013 18:01:08 GMT+1200 (New Zealand Standard Time)	-41.6327	174.2831	21	3.4	Google Map	OSM Map
Tue Jul 23 2013 18:05:20 GMT+1200 (New Zealand Standard Time)	-38.8079	175.4547	140	2.1	Google Map	OSM Map
Tue Jul 23 2013 18:11:13 GMT+1200 (New Zealand Standard Time)	-41.6919	174.3964	8	2.3	Google Map	OSM Map
Tue Jul 23 2013 18:16:41 GMT+1200 (New Zealand Standard Time)	-41.7113	174.1772	18	2.7	Google Map	OSM Map
Tue Jul 23 2013 18:23:04 GMT+1200 (New Zealand Standard Time)	-41.6946	174.0623	5	1.9	Google Map	OSM Map
Tue Jul 23 2013 18:25:52 GMT+1200 (New Zealand Standard Time)	-41.6837	174.2292	11	4.3	Google Map	OSM Map
Tue Jul 23 2013 18:26:30 GMT+1200 (New Zealand Standard Time)	-40.0264	175.5516	9	3.5	Google Map	OSM Map
Tue Jul 23 2013 18:38:02 GMT+1200 (New Zealand Standard Time)	-41.5181	174.414	12	1.8	Google Map	OSM Map
Tue Jul 23 2013 18:41:11 GMT+1200 (New Zealand Standard Time)	-41.5313	174.4298	8	2.2	Google Map	OSM Map

Data Table *with* Bootstrap Styling

Without the styling it would look like this;

DTG	Lat	Long	Depth	Magnitude	Google Map	OSM Map
Earthquake Table						
Tue Jul 23 2013 17:50:01 GMT+1200 (New Zealand Standard Time)	-41.6244	174.3607	10	2.5	Google Map	OSM Map
Tue Jul 23 2013 18:01:08 GMT+1200 (New Zealand Standard Time)	-41.6327	174.2831	21	3.4	Google Map	OSM Map
Tue Jul 23 2013 18:05:20 GMT+1200 (New Zealand Standard Time)	-38.8079	175.4547	140	2.1	Google Map	OSM Map
Tue Jul 23 2013 18:11:13 GMT+1200 (New Zealand Standard Time)	-41.6919	174.3964	8	2.3	Google Map	OSM Map
Tue Jul 23 2013 18:16:41 GMT+1200 (New Zealand Standard Time)	-41.7113	174.1772	18	2.7	Google Map	OSM Map
Tue Jul 23 2013 18:23:04 GMT+1200 (New Zealand Standard Time)	-41.6946	174.0623	5	1.9	Google Map	OSM Map
Tue Jul 23 2013 18:25:52 GMT+1200 (New Zealand Standard Time)	-41.6837	174.2292	11	4.3	Google Map	OSM Map
Tue Jul 23 2013 18:26:30 GMT+1200 (New Zealand Standard Time)	-40.0264	175.5516	9	3.5	Google Map	OSM Map
Tue Jul 23 2013 18:38:02 GMT+1200 (New Zealand Standard Time)	-41.5181	174.414	12	1.8	Google Map	OSM Map
Tue Jul 23 2013 18:41:11 GMT+1200 (New Zealand Standard Time)	-41.5313	174.4298	8	2.2	Google Map	OSM Map

Data Table *without* Bootstrap Styling

We will be adding to this grid layout section as we add in charts which will want their own allocated space on our page.

The next section of the file starts our JavaScript and declares our variables for our charts.

```
// Create the dc.js chart objects & link to div
var dataTable = dc.dataTable("#dc-table-graph");
```

The first line assigns the variable `dataTable` to the dc.js dataTable chart type (`var dataTable = dc.dataTable("#dc-table-graph");`) and assigns the chart to the ID selector `dc-table-graph`.

Then we get into the d3.js.

```
// load data from a csv file
d3.csv("quakes.csv", function (data) {

  // format our data
  var dtgFormat = d3.time.format("%Y-%m-%dT%H:%M:%S");

  data.forEach(function(d) {
    d.dtg   = dtgFormat.parse(d.origintime.substr(0,19));
    d.lat   = +d.latitude;
    d.long  = +d.longitude;
    d.mag   = d3.round(+d.magnitude,1);
    d.depth = d3.round(+d.depth,0);
  });
```

We load our csv file with the line `d3.csv("quakes.csv", function (data) {`. I have deliberately left this file in its raw form as received from Geonet. Its format looks a little like this (be warned, the formatting of the book can create word wrap issues where the text will be broken by a backslash () and this is likely to happen with the text below);

```
FID,publicid,origintime,longitude,latitude,depth,magnitude,magnitudetype,status,p\
hases,type,agency,updatetime,origin_geom
quake.2013p550753,2013p550753,2013-07-23T18:41:11.707,174.4298,-41.5313,7.9883,2.\
2425,M,automatic,27,,WEL(GNS_Primary),2013-07-23T18:43:15.672,POINT (174.42978 -4\
1.531299)
quake.2013p550747,2013p550747,2013-07-23T18:38:02.481,174.414,-41.5181,11.6797,1.\
7892,M,automatic,11,,WEL(GNS_Primary),2013-07-23T18:39:25.37,POINT (174.41398 -41\
.518114)
quake.2013p550725,2013p550725,2013-07-23T18:26:30.229,175.5516,-40.0264,8.75,3.45\
62,M,automatic,21,,WEL(GNS_Primary),2013-07-23T18:29:46.305,POINT (175.55155 -40.\
026412)
```

We then declare a small function that will format our time correctly (`var dtgFormat = d3.time.format("%Y-%m-%dT%H:%M:%S");`). This follows exactly the same procedure we took when creating our very first simple line graph at the start of the book.

However, there is a slight twist... Observant readers will notice that while we have a function that resolves a date/time that is formatted with year, month, day, hour, minute and second values, I don't include an allowance for the fractions of seconds that appear in the csv file. Well spotted. The reason for this is that in spite of initially including this formatting, I found it caused some behaviour that I couldn't explain, so I reverted to ***cheating*** and you will note that in the next section when I format the values from the csv file, I truncate the date/time value to the first 19 characters (`d.origintime.substr(0,19)`). This solved my problem by chopping off the fractions of a second (admittedly without actually solving the underlying issue) and I moved on with my life.

While we're on the subject, observant readers will have noticed that the format of the date / time that appears in the table are (how to put this kindly.....), not what came out of the csv file.

If you want to put this in a different format we can employ the same technique we used when formatting time figures in the section that dealt with tables. All we need to do is to assign a new variable for our 'correctly' formatted time in the `forEach` loop. and then call that variable when displaying the table values.

The following code will create a date / time string in the format yyyy-mm-dd hh:mm:ss with a variable name dtg1 (put this in the `forEach` loop).

```
d.dtg1  = d.origintime.substr(0,10) + " " + d.origintime.substr(11,8);
```

Then, when your code calls the values for the table, instead of the line that says;

```
      function(d) { return d.dtg; },
```

You rename `dtg` to `dtg1` like so;

```
      function(d) { return d.dtg1; },
```

The end result will look like this;

DTG	Lat	Long	Depth	Magnitude	Google Map	OSM Map
Earthquake Table						
2013-07-23 17:50:01	-41.6244	174.3607	10	2.5	Google Map	OSM Map
2013-07-23 18:01:08	-41.6327	174.2831	21	3.4	Google Map	OSM Map
2013-07-23 18:05:20	-38.8079	175.4547	140	2.1	Google Map	OSM Map
2013-07-23 18:11:13	-41.6919	174.3964	8	2.3	Google Map	OSM Map
2013-07-23 18:16:41	-41.7113	174.1772	18	2.7	Google Map	OSM Map
2013-07-23 18:23:04	-41.6946	174.0623	5	1.9	Google Map	OSM Map

Data Table with Formatted Date / Time

As mentioned, the next section goes through each of the records and formats them correctly. The date/time gets formatted, the latitude and longitude are declared as numerical values (if they weren't already) and the magnitude and depth values are rounded to make the process of grouping them simpler.

```
data.forEach(function(d) {
  d.dtg   = dtgFormat.parse(d.origintime.substr(0,19));
  d.lat   = +d.latitude;
  d.long  = +d.longitude;
  d.mag   = d3.round(+d.magnitude,1);
  d.depth = d3.round(+d.depth,0);
});
```

The next section in our code sets up the dimensions and groupings for the dc.js chart type and crossfilter functions.

```
// Run the data through crossfilter and load our 'facts'
var facts = crossfilter(data);

// Create dataTable dimension
var timeDimension = facts.dimension(function (d) {
  return d.dtg;
});
```

We load all of our data into crossfilter (`var facts = crossfilter(data);`) and give it the name `facts`.

Then we create a dimension from our data (`facts`) of the date/time values.

```
var timeDimension = facts.dimension(function (d) {
  return d.dtg;
});
```

The last major chunk of code is the piece that configures our data table.

```
dataTable.width(960).height(800)
  .dimension(timeDimension)
      .group(function(d) { return "Earthquake Table"
       })
      .size(10)
  .columns([
    function(d) { return d.dtg; },
    function(d) { return d.lat; },
    function(d) { return d.long; },
    function(d) { return d.depth; },
    function(d) { return d.mag; },
    function(d) {
        return '<a href=\"http://maps.google.com/maps?z=12&t=m&q=loc:' +
            d.lat + '+' + d.long +"\" target=\"_blank\">Google Map</a>"},
    function(d) {
        return '<a href=\"http://www.openstreetmap.org/?mlat=' +
```

```
            d.lat + '&mlon=' + d.long +
            '&zoom=12'+ "\" target=\"_blank\"> OSM Map</a>"}
    ])
    .sortBy(function(d){ return d.dtg; })
    .order(d3.ascending);
```

Firstly the width and height are declared (`dataTable.width(960).height(800)`). Then the dimension of the data that will be used is declared (`.dimension(timeDimension)`).

Separate sections of the table can have a header applied. In this case the entire table is given the grouping 'Earthquake Table' (`.group(function(d) { return "Earthquake Table"})`), but several examples online give the date.

The `.size(10)` line sets the maximum number of lines of the table to be displayed to 10.

Then we have the block of code that sets what data appears in which columns. It should be noted that this matches up with the headers that were declared in the earlier section of the code where the divs for the table were laid out.

The portion of this block that has a '*little bit of fancy*' are the two columns that set links that allow a user to click on the designation 'Google Map' or 'OSM Map' and have the browser open a new window containing a Google or Open Street Map (OSM) map with a marker designating the location of the quake. I won't mention too much about how the links are made up other than to say that they are pretty much a combination of the latitude, longitude and zoom level for both. Please check out the code for more.

Lastly we sort by the date/time value (`.sortBy(function(d){ return d.dtg; })`) in ascending order (`.order(d3.ascending);`).

The final part of our JavaScript renders all our charts (`dc.renderAll();`) and then closes off the initial `d3.csv` call.

```
  // Render the Charts
  dc.renderAll();

});
```

The final part of our code simply closes off the `<script>`, `<body>` and `<html>` tags.

There we have it. The template for starting to play with different crossfiltered dc.js charts.

Add a Bar Chart.

The ubiquitous bar chart is a smart choice if you're starting out with crossfilter and dc.js. It's pretty easy to implement and gives a certain degree of instant satisfaction.

The bar chart that we'll create will be a representation of the magnitude of the earthquakes that we have in our dataset. In this respect, what we are expecting to see is the magnitude of the events along the x axis and the number of each such event on the y axis.

It should end up looking a bit like this.

Bar Chart Example

We'll work through adding the chart in stages (and this should work for subsequent charts). Firstly we'll organise a position for our chart on the page using the bootstrap grid set-up. Then we'll name our chart and assign it a chart type. Then we'll create any required dimension and grouping and finally we'll configure the parameters for the chart. Sounds simple right?

1. Position the chart
2. Assign type
3. Dimension and Group
4. Configure chart parameters

Position the bar chart

We are going to position our bar chart above our data table and we'll actually only make it half the width of our data table so that we can add in another one along side it later.

Just under the line of code that defined the main container for the layout;

```
<div class='container' style='font: 12px sans-serif;'>
```

We add in a new row that has two `span6`'s in it (remembering our total is a span of 12 (see the section on bootstrap layout if it's a bit unfamiliar)).

```
<div class='row'>
  <div class='span6' id='dc-magnitude-chart'>
    <h4>Events by Magnitude</h4>
  </div>
  <div class='span6' id='blank'>
    <h4>Blank</h4>
  </div>
</div>
```

We've given the first `span6` an ID selector of `dc-magnitude-chart`. So when we we assign our chart that selector, it will automatically appear in that position. We've also put a simple title in place (`<h4>Events by Magnitude</h4>`). The second `span6` is set as blank for the time being (we'll put another bar chart in it later).

Assign the bar chart type

Here we give our chart it's name (`magnitudeChart`), assign it with a dc.js chart type (in this case `barChart`) and assign it to the ID selector (`dc-magnitude-chart`).

Under the line that assigns the `dataTable` chart type...

```
var dataTable = dc.dataTable("#dc-table-graph");
```

... add in the equivalent for our bar chart.

```
var dataTable = dc.dataTable("#dc-table-graph");
var magnitudeChart = dc.barChart("#dc-magnitude-chart");
```

All done.

Dimension and group the bar chart data

To set our dimension for magnitude, it's as simple as following the same format as we had previously done for the data table but in this case using the `.mag` variable.

This should go just before the portion of the code that created the data table dimension.

```
var magValue = facts.dimension(function (d) {
  return d.mag;
});
```

This dimension (`magValue`) has been set and now has, as its index, each unique magnitude that is seen in the database. This is essentially defining the values on the x axis for our bar chart.

Then we want to group the data by counting the number of events of each magnitude.

```
var magValueGroupCount = magValue.group()
  .reduceCount(function(d) { return d.mag; }) // counts
```

This piece of code (which should go directly under the `magValue` dimension portion), groups (`.group()`) by counting (`.reduceCount`) all of the magnitude values (`function(d) { return d.mag; }`) and assigns it to the `magValueGroupCount` variable. This has essentially defined the values for the y axis of our bar chart (the number of times each magnitude occurs).

Configure the bar chart parameters

There are lots of parameters that can be configured, and if the truth be told, I haven't explored all of them or, in some cases, worked out exactly how they work.

However, the best way to learn is by doing, so here is the block of code for configuring the bar chart. This should go just before the block that configures the dataTable.

```
magnitudeChart.width(480)
  .height(150)
  .margins({top: 10, right: 10, bottom: 20, left: 40})
  .dimension(magValue)
  .group(magValueGroupCount)
  .transitionDuration(500)
  .centerBar(true)
  .gap(65)
  .filter([3, 5])
  .x(d3.scale.linear().domain([0.5, 7.5]))
  .elasticY(true)
  .xAxis().tickFormat();
```

That should be it. With the addition of this portion of the code, you should have a functioning visualization that can be filtered dynamically. Just check to make sure that everything is working properly and we'll go through some of the configuration options to see what they do.

Your web page should look a little like this;

Web Page with Bar Chart

The configuration options start by declaring the name of the chart (`magnitudeChart`) and setting the height and width of the chart.

```
magnitudeChart.width(480)
  .height(150)
```

In the case of our example I have selected the width based on the default size for a `span6` grid segment in bootstrap and adjusted the height to make it look suitable.

Then we have our margins set up.

```
  .margins({top: 10, right: 10, bottom: 20, left: 40})
```

Nothing too surprising there although the left margin is slightly larger to allow for larger values on the y axis to be represented without them getting clipped.

Then we define which dimension and grouping we will use.

```
  .dimension(magValue)
  .group(magValueGroupCount)
```

I like to think of this section as the `.dimension` declaration being the x axis and the `.group` declaration being the y axis. This just helps me get the graph straight in my head before it's plotted.

The `.transitionDuration` setting defines the length of time that any change takes to be applied to the chart as it adjusts.

```
      .transitionDuration(500)
```

Then we ensure that the bar for the bar graph is centred on the ticks on the x axis.

```
.centerBar(true)
```

Without this (`true` is not the default), the graph will look slightly odd.

Bar Chart with Bars Not Centred

The setting of the gap between the bars is accomplished with the following setting;

```
.gap(65)
```

I will admit that I still don't quite understand how this setting works exactly, but I can get it to do what I want with a little trial and error.

For instance, I would expect that `.gap(2)` would have the effect of producing a gap of 2 pixels between the bars. But this would be the result for our graph if I have that set.

Bar Chart with `gap` Set to 2

If you select a portion of the graph you will see some strange things going on. That appears to be as a result of the bars being too wide for the graph.

Setting the gap for a bar graph is a pretty tricky thing to do (programmatically), and I can see why it would throw some strange results. The way around this and the way to find the ideal `.gap` setting is to set the `.gap` value high and then reduce it till it's right.

For instance, if we set it to 100 (`.gap(100)`) we will get the following result.

Bar Chart with gap Set to 100

Then we just keep backing the values off till we reach an acceptable chart on the screen.

In the case of our example, it's `.gap(65)`.

Bar Chart Example

I have added in the next setting more because I want you to know it exists, rather than wanting to use it in this example.

```
.filter([3, 5])
```

Bar Chart with Pre-Selection Section

Setting the `.filter` configuration will load the graph with a portion of it pre-selected. If you omit this parameter, the entire graph is selected by default. In most cases that I can think of, that is what I would start with.

We can set the range of values presented in our graph by defining the domain (in the same way as for d3.js).

```
.x(d3.scale.linear().domain([0.5, 7.5]))
```

The next parameter sets the y axis to adjust dynamically as the filtered data is returned.

```
        .elasticY(true)
```

The final parameter that we set is to format the values on the x axis.

```
        .xAxis().tickFormat();
```

And that's it! A bar graph added to your visualization with full dynamic control.

Just one more thing...

Just another snippet that could be useful. In the section where we set up our group to count the number of instances of individual magnitudes we had;

```
var magValueGroupCount = magValue.group()
  .reduceCount(function(d) { return d.mag; }) // counts
```

We could have just as easily summed the magnitude values instead of counting them by using `.reduceSum` instead of `.reduceCount`. This has the effect of increasing the value on the y axis (as the sum of the magnitudes would have been greater than the count) like so

Bar Chart Counting and Summing

The reason I mention it is that *summing* the numeric value would be useful in many circumstances (file size or packet size or similar).

Just yet another thing...

When we initially set up our grid layout for the web page we left ourselves a blank position for another graph. If you feel so inclined, try to include another bar graph in this position that will display the depth of the earthquakes.

The example I came up with looks like this;

Earthquake page with Magnitude and Depth Bar Charts

And the sections I added are as follows;

Position the chart

(more of a change than an addition)

```
<div class='span6' id='dc-depth-chart'>
      <h4>Events by Depth (km)</h4>
</div>
```

Assign type

```
var depthChart = dc.barChart("#dc-depth-chart");
```

Dimension and Group

```
var depthValue = facts.dimension(function (d) {
  return d.depth;
});
var depthValueGroup = depthValue.group();
```

Configure chart parameters

```
depthChart.width(480)
  .height(150)
  .margins({top: 10, right: 10, bottom: 20, left: 40})
  .dimension(depthValue)
  .group(depthValueGroup)
  .transitionDuration(500)
  .centerBar(true)
  .gap(1)
  .x(d3.scale.linear().domain([0, 100]))
  .elasticY(true)
  .xAxis().tickFormat(function(v) {return v;});
```

Add a Line Chart.

The line chart is another simple choice for implementation using crossfilter and dc.js.

The line chart that we'll create will be a representation of the frequency of the occurrence of the earthquakes that we have in our dataset. In this respect, what we are expecting to see is the number of events on the y axis and the time-scale on the x axis.

It should end up looking a bit like this.

Line Chart Example

Just as with the bar chart, we'll work through adding the chart in the following stages.

1. Position the chart
2. Assign type
3. Dimension and Group
4. Configure chart parameters

Position the line chart

We are going to position our line chart above our data table (and below the bar charts) and we'll make it the full width of our data table so that it looks like it belongs there.

Just under the line of code that defined the containers for the bar graphs;

```
<div class='row'>
  <div class='span6' id='dc-magnitude-chart'>
    <h4>Events by Magnitude Counted</h4>
  </div>
  <div class='span6' id='dc-depth-chart'>
        <h4>Events by Depth (km)</h4>
  </div>
</div>
```

We add in a new row that has a single `span12`.

```
<div class='row'>
  <div class='span12' id='dc-time-chart'>
    <h4>Events per hour</h4>
  </div>
</div>
```

We've given it an ID selector of `dc-time-chart`. So when we assign our chart that selector, it will automatically appear in that position. We've also put another simple title in place (`<h4>Events per hour</h4>`).

Assign the line chart type

Here we give our chart it's name (`timeChart`), assign it with a dc.js chart type (in this case `lineChart`) and assign it to the ID selector (`dc-time-chart`).

Under the line that assigns the `depthChart` chart type...

```
var depthChart = dc.barChart("#dc-depth-chart");
```

... add in the equivalent for our line chart.

```
var depthChart = dc.barChart("#dc-depth-chart");
var timeChart = dc.lineChart("#dc-time-chart");
```

Nice.

Dimension and group the line chart data

We'll put the code between the dimension and group of the depth chart and the data table dimension (this is just to try and keep the code in the same order as the graphs on the page).

To set our dimension for our time we do something a little different.

```
var volumeByHour = facts.dimension(function(d) {
  return d3.time.hour(d.dtg);
});
```

This dimension (`volumeByHour`) uses the same `facts` data, but when the key values are returned (`return d3.time.hour(d.dtg);`) we are going to return the information by hours. This is essentially defining the resolution of the values on the x axis for our line chart.

Then we want to group the data by counting the number of events of for each hour.

```
var volumeByHourGroup = volumeByHour.group()
  .reduceCount(function(d) { return d.dtg; });
```

This piece of code (which should go directly under the `volumeByHour` dimension portion) groups (`.group()`) by counting (`.reduceCount`) all of the magnitude values (`function(d) { return d.dtg; }`) and assigns it to the `volumeByHourGroup` variable. This has defined the values for the y axis of our line chart (the number of events we see in a given hour).

Configure the line chart parameters

As with the bar chart, there are lots of parameters that can be configured. The best way to learn what they do is by having a play with them. So here is the block of code for configuring the line chart. Once you are happy that it works on your system, take some time and go through the settings in conjunction with the information from the demo page[365] and the api reference[366].

This should go just before the block that configures the dataTable (again, this is just to try and keep the code in the same order as the graphs on the page).

```
// time graph
timeChart.width(960)
  .height(150)
  .margins({top: 10, right: 10, bottom: 20, left: 40})
  .dimension(volumeByHour)
  .group(volumeByHourGroup)
  .transitionDuration(500)
  .elasticY(true)
  .x(d3.time.scale().domain([new Date(2013, 6, 18), new Date(2013, 6, 24)]))
  .xAxis();
```

That should be it. With the addition of this portion of the code, you should have a functioning visualization that can be filtered dynamically. Just check to make sure that everything is working properly and we'll go through some of the configuration options to see what they do.

To start with, your page should look something like this;

[365] http://nickqizhu.github.io/dc.js/
[366] https://github.com/NickQiZhu/dc.js/wiki/API

Web Page with Line Chart

The configuration options start by declaring the name of the chart (`timeChart`) and setting the height and width of the chart.

```
timeChart.width(960)
  .height(150)
```

In the case of our example I have selected the width based on the default size for a `span12` grid segment in bootstrap and adjusted the height to make it look suitable.

Then we have our margins set up.

```
  .margins({top: 10, right: 10, bottom: 20, left: 40})
```

Nothing too surprising there although the left margin is slightly larger to allow for larger values on the y axis to be represented without them getting clipped (not strictly for this example, but it's a handy default).

Then we define which dimension and grouping we will use.

```
  .dimension(volumeByHour)
  .group(volumeByHourGroup)
```

Think of the `.dimension` declaration being the x axis and the `.group` declaration being the y axis.

The `.transitionDuration` setting defines the length of time that any change takes to be applied to the chart as it adjusts.

```
        .transitionDuration(500)
```

We can set the y axis to dynamically adjust when the number of events are filtered by selections on any of the other charts.

```
    .elasticY(true)
```

For instance if we select only earthquakes with a magnitude between 4 and 5, our line chart will have a maximum value on the y axis of 7 events;

Line Chart y Axis Low

However, if we select all the earthquakes, the y axis will dynamically adjust to over 30.

Line Chart y Axis High

Since the line chart has an x axis which is made of date/time values, we set our scale and domain using the `d3.time.scale` declaration.

```
    .x(d3.time.scale().domain([new Date(2013, 6, 18), new Date(2013, 6, 24)]))
```

This is hard coded for our date range, but a smarter method would be to have the scale adjust to suit your range of date/time values automatically with the following line;

```
    .x(d3.time.scale().domain(d3.extent(data, function(d) { return d.dtg; })))
```

Using the `d3.extent` function means that our line graph of time now spans the exact range of our data values on the x axis (note that the time scale now starts just before the 18th and ends when our data ends).

Line Chart with Better x Axis

The final parameter that we set is to add the x axis.

```
        .xAxis();
```

Adding tooltips to a line chart

dc.js has a nice feature for adding tooltips to a line chart.

It utilises the `.title` function in the configuration of the chart to apply the tooltip, but the downside is that the ability to select the time range needs to be disabled (there are ways to compensate for this which I hope to cover in the future).

If we take our example line chart configuration block of code;

```
// time graph
timeChart.width(960)
  .height(150)
  .margins({top: 10, right: 10, bottom: 20, left: 40})
  .dimension(volumeByHour)
  .group(volumeByHourGroup)
  .transitionDuration(500)
  .elasticY(true)
  .x(d3.time.scale().domain([new Date(2013, 6, 18), new Date(2013, 6, 24)]))
  .xAxis();
```

We need to turn off the `.brushOn` feature (`.brushOn(false)`) that allows for selection and add in the `.title` function as follows;

```
// time graph
timeChart.width(960)
  .height(150)
  .margins({top: 10, right: 10, bottom: 20, left: 40})
  .dimension(volumeByHour)
  .group(volumeByHourGroup)
  .transitionDuration(500)
  .brushOn(false)
  .title(function(d){
    return d.data.key
    + "\nNumber of Events: " + d.data.value;
    })
  .elasticY(true)
  .x(d3.time.scale().domain([new Date(2013, 6, 18), new Date(2013, 6, 24)]))
  .xAxis();
```

Line Chart with Tooltip

As we can see, the tooltip is using the default time format for the script from our `key` value (on the x axis), and as a result, the representation of the date / time is quite long winded. We can adapt this to a format of our choosing by calling a time formatting function similar to the following;

```
var dtgFormat2 = d3.time.format("%a %e %b %H:%M");
```

This line could ideally go after the other time formatting function (`dtgFormat`) that occurs earlier in the script. The formatting it's introducing can be found in the d3.js wiki[367], but in short it returns the date / time formatted as abbreviated weekday name, day of the month as a decimal number, abbreviated month name and 24 hour clock hour:minute.

With our function in place, the `.title.` call from our line chart configuration code would now look like this;

[367] https://github.com/mbostock/d3/wiki/Time-Formatting#wiki-format

```
.title(function(d){
  return dtgFormat2(d.data.key)
  + "\nNumber of Events: " + d.data.value;
  })
```

And the resulting graph looks like this;

Line Chart with Improved Tooltip

We also add in the number of the events from the y axis (`d.data.value`), separated with a new line character (`\n`) and some appropriate text.

Add a Row Chart.

The row chart provides an excellent mechanism for presenting and filtering on discrete values or identifiers.

The row chart that we'll create will be a representation of the number of earthquake events that occur on a particular day of the week. As such it doesn't represent any logical reason for selecting a Saturday over a Wednesday, and it is used here solely because the data makes a nice row chart :-). In this respect, what we are expecting to see is the number of events on the x axis and the individual days on the y axis.

It should end up looking a bit like this.

Row Chart Example

Now for a super cool feature with row charts...

Click on one of the rows...

Selecting a Row

How about that!

You can select an individual row from your chart and all the other rows reflect the selection.

Go ahead and select other combinations of more than one row if you want. Welcome to data immersion!

Just as with the previous chart examples, we'll work through adding the chart in the following stages.

1. Position the chart
2. Assign type
3. Dimension and Group
4. Configure chart parameters

Position the row chart

We are going to position our row chart above our data table (and below the line chart) and we'll divide the row that it sits in into 3 equally spaced spans of `span3`. The additional two spans we'll leave blank for future use.

Just under the row of code that defined the containers for the line graph;

```
<div class='row'>
  <div class='span12' id='dc-time-chart'>
    <h4>Events per hour</h4>
  </div>
</div>
```

We add in a new row that has our three `span4`'s.

```
<div class='row'>
  <div class='span4' id='dc-dayweek-chart'>
    <h4>Day of the Week</h4>
  </div>
  <div class='span4' id='blank1'>
        <h4>Blank 1</h4>
  </div>
  <div class='span4' id='blank2'>
        <h4>Blank 2</h4>
  </div>
</div>
```

We've given it an ID selector of `dc-dayweek-chart`. So when we assign our chart that selector, it will automatically appear in that position. We've also put another simple title in place (`<h4>Day of the Week</h4>`).

The additional two `span4s` have been left blank.

Assign the row chart type

Here we give our chart its name (`dayOfWeekChart`), assign it with a dc.js chart type (in this case `rowChart`) and assign it to the ID selector (`dc-dayweek-chart`).

Under the row that assigns the `depthChart` chart...

```
var depthChart = dc.barChart("#dc-depth-chart");
```

... add in the equivalent for our row chart.

```
var dayOfWeekChart = dc.rowChart("#dc-dayweek-chart");
```

Dimension and group the row chart data

We'll put the code between the dimension and group of the line (time) chart and the data table dimension (this is just to try and keep the code in the same order as the graphs on the page).

When adding our dimension for our day of the week we want to provide an appropriate label so our code does something extra.

```
var dayOfWeek = facts.dimension(function (d) {
  var day = d.dtg.getDay();
  switch (day) {
    case 0:
      return "0.Sun";
    case 1:
      return "1.Mon";
    case 2:
      return "2.Tue";
    case 3:
      return "3.Wed";
    case 4:
      return "4.Thu";
    case 5:
      return "5.Fri";
    case 6:
      return "6.Sat";
  }
});
```

This dimension (`dayOfWeek`) uses the same `facts` data, but when we return our key values we are going to return them as a combination of their numerical order (0 = Sunday etc) and their abbreviation (Sun = Sunday etc). This is essentially defining the categories of the values on the y axis for our row chart.

The code snippet looks a little strange, but think of it as extracting the numerical representation of the day of the week from our data (`var day = d.dtg.getDay();`) and then matching each number with an appropriate label (0 = '0.Sun', 1 = '1.Mon' etc). It's these labels that are now our key values in our dimension.

Then we want to group the data by using the default action of the `.group()` function to count the number of events for each day of the week.

```
var dayOfWeekGroup = dayOfWeek.group();
```

Configure the row chart parameters

As with the previous charts, there are plenty of parameters that can be configured. The best way to learn what they do is still to have a play with them. So here is the block of code for configuring the row chart. Once you are happy that it works on your system, take some time and go through the settings in conjunction with the information from the demo page[368] and the api reference[369].

This should go just before the block that configures the dataTable (again, this is just to try and keep the code in the same order as the graphs on the page).

```
// row chart day of week
dayOfWeekChart.width(300)
  .height(220)
  .margins({top: 5, left: 10, right: 10, bottom: 20})
  .dimension(dayOfWeek)
  .group(dayOfWeekGroup)
  .colors(d3.scale.category10())
  .label(function (d){
     return d.key.split(".")[1];
  })
  .title(function(d){return d.value;})
  .elasticX(true)
  .xAxis().ticks(4);
```

That should get you working. With the addition of this portion of the code, you should have a functioning visualization that can be filtered dynamically by clicking on the appropriate day of the week in your row chart. Just check to make sure that everything is working properly and we'll go through some of the configuration options to see what they do.

To start with, your page should look something like this;

[368] http://nickqizhu.github.io/dc.js/
[369] https://github.com/NickQiZhu/dc.js/wiki/API

Web Page with Row Chart

The configuration options start by declaring the name of the chart (`dayOfWeekChart`) and setting the height and width of the chart.

```
dayOfWeekChart.width(300)
  .height(220)
```

In the case of our example I have selected the width based on the default size for a `span4` grid segment in bootstrap and adjusted the height to make it look suitable.

Then we have our margins set up.

```
  .margins({top: 5, left: 10, right: 10, bottom: 20})
```

Nothing too surprising there although I did reduce the top margin slightly more than I thought I would need. You can be the judge for your own charts.

Then we define which dimension and grouping we will use.

```
.dimension(dayOfWeek)
.group(dayOfWeekGroup)
```

For a row chart, think of the `.dimension` declaration being the y axis and the `.group` declaration being the x axis (the opposite to the previous charts).

We can set the range of colours to use one of the standard palettes[370].

```
.colors(d3.scale.category10())
```

Then we add the labels to our categories by splitting the key values (remember `0.Sun`, `1.Mon` etc) at the decimal point and returning the second part of the split value (which is the `Sun`, `Mon` part) as the label.

```
.label(function (d){
   return d.key.split(".")[1];
 })
```

A cool way to prove this is to change the variable that returns the label to use the 1st part of the split value buy using a `[0]` instead of a `[1]` with code like this;

```
.label(function (d){
   return d.key.split(".")[0];
 })
```

The end result produces...

Row Chart with the First Part of the Key Value

The next line in the configuration adds a tool tip to our row chart using the value when the mouse hovers over the appropriate bar.

```
.title(function(d){return d.value;})
```

[370] http://www.schneidy.com/Tutorials/ColorTutorial.html

Row Chart Tool Tip

We can set the x axis to dynamically adjust when the number of events are filtered by selections on any of the other charts using the following configuration line.

```
.elasticX(true)
```

For instance if we select a subset of the earthquakes using our time / line chart, our row chart will have a corresponding selection of the appropriate days and the x axis will alter accordingly.

Selection Effect on Row Chart and Dynamic X Axis

Lastly we set up our x axis with 4 ticks.

```
.xAxis().ticks(4);
```

Add a Pie Chart.

The pie chart provides an useful way of presenting and filtering on discrete values or identifiers similar to a row chart.

The pie chart that we'll create will be a representation of which island the earthquakes occurred in. For those of you unfamiliar with the stunning landscape of New Zealand, there are two main islands creatively named North Island and South Island (stunning *and* practical!). The determination of what constitutes the North and South Island has been decided in a completely unscientific way (by me) by designating any area South of latitude -40.555907 and West of longitude 174.590607 as the South Island and anything else is the North Island.

Determination of North and South

The pie graph should end up looking a bit like this.

Pie Chart Example

Good news! The pie chart shares the same cool feature as the row chart...

Click on one of the pie segments...

Selecting a Pie Segment

... and everything dynamically reflects the selection.

Just as with the previous chart examples, we'll work through adding the chart in the following stages.

1. Position the chart
2. Assign type
3. Dimension and Group
4. Configure chart parameters

Position the pie chart

We are going to position our pie chart above our data table (and below the line chart) in the same row as the row chart in one of the blank `span4`'s.

The code that sets up that row should now look like this;

```
<div class='row'>
  <div class='span4' id='dc-dayweek-chart'>
    <h4>Day of the Week</h4>
  </div>
  <div class='span4' id='dc-island-chart'>
        <h4>North or South Island</h4>
  </div>
  <div class='span4' id='blank2'>
        <h4>Blank 2</h4>
  </div>
</div>
```

We've given it an ID selector of `dc-island-chart`. So when we assign our chart that selector, it will automatically appear in that position. We've also put another simple title in place (`<h4>North or South Island</h4>`).

The last `span4` is still blank.

Assign the pie chart type

Here we give our chart its name (`dayOfWeekChart`), assign it with a dc.js chart type (in this case `pieChart`) and assign it to the ID selector (`dc-dayweek-chart`).

Under the row that assigns the `dayOfWeekChart` chart...

```
var dayOfWeekChart = dc.rowChart("#dc-dayweek-chart");
```

... add in the equivalent for our pie chart.

```
var islandChart = dc.pieChart("#dc-island-chart");
```

Dimension and group the pie chart data

We'll put the code between the dimension and group of the row chart and the data table dimension (this is just to try and keep the code in the same order as the graphs on the page).

When adding our dimension for our islands we want to provide an appropriate label so our code does the figuring out based on the latitude and longitude that we had established as the boundary between North and South.

```
var islands = facts.dimension(function (d) {
  if (d.lat <= -40.555907 && d.long <= 174.590607)
    return "South";
  else
    return "North";
  });
```

This dimension (`islands`) uses the same `facts` data, but when we return our key values we are going to return them as either 'North' or 'South'. To do this we employ a simple `if` statement with a little logic. These are the only two 'slices' for our pie chart.

Then we want to group the data by using the default action of the `.group()` function to count the number of events for each day of the week.

```
var islandsGroup = islands.group();
```

Configure the pie chart parameters

There are fewer parameters that can be configured for pie charts, but we'll still take the time to go through the options used here.

This code should go just before the block that configures the dataTable (again, this is just to try and keep everything in the same order as the graphs on the page).

```
islandChart.width(250)
  .height(220)
  .radius(100)
  .innerRadius(30)
  .dimension(islands)
  .group(islandsGroup)
  .title(function(d){return d.value;});
```

That should get the chart working. With the addition of this portion of the code, you should have a functioning visualization that can be filtered dynamically by clicking on the appropriate island in your pie chart. Just check to make sure that everything is working properly and we'll go through some of the configuration options to see what they do.

To start with, your page should look something like this;

Web Page with Pie Chart

The configuration options start by declaring the name of the chart (`islandChart`) and setting the height and width of the chart.

```
islandChart.width(250)
  .height(220)
```

In the case of our example I have selected the width based on the default size for a `span4` grid segment in bootstrap and adjusted the height to make it look suitable alongside the row chart.

Then we set up our inner and outer radii for our pie.

```
  .radius(100)
  .innerRadius(30)
```

This is fairly self explanatory, but by all means adjust away to make sure the chart suits your visualization.

Then we define which dimension and grouping we will use.

```
.dimension(islands)
.group(islandsGroup)
```

For a pie chart, the `.dimension` declaration is the discrete values that make up each segment of the pie and the `.group` declaration is the size of the pie.

The final line in the configuration adds a tool tip to our pie chart using the value when the mouse hovers over the appropriate slice.

```
.title(function(d){return d.value;})
```

Pie Chart Tool Tip

Resetting filters

Once you have made selections on some of your data dimensions, often you will want to reset those selections to return to a stable state.

For example, when selecting different days to display in the row chart, if you have three days selected as so…

Selected Elements in Row Chart

… to return to the default setting where all the days are selected can be a bit of a pain.

Instead, we can use a dc.js 'reset' feature where a 'reset' label is generated to allow us revert to the starting condition.

There is a simple way to enable this feature, but we'll take an additional few steps to make it look slightly better (and to learn some new tricks).

In the simplest method, this feature simply involves adding in the following code to the section where we add in the `rows` and `spans` when setting out our layout.

```
<a class="reset"
  href="javascript:dayOfWeekChart.filterAll();dc.redrawAll();"
  style="display: none;">
  reset
</a>
```

In the case of our example row chart, that would then look a bit like this;

```
<div class='span4' id='dc-dayweek-chart'>
  <h4>Day of the Week</h4>
  <a class="reset"
    href="javascript:dayOfWeekChart.filterAll();dc.redrawAll();"
    style="display: none;">
    reset
  </a>
</div>
```

The additional code adds in a link (that's the `<a>` tags) with a specific class that designates its function (the `class="reset"` part (this is what will let dc.js know what to do)). The link action (`href="javascript:dayOfWeekChart.filterAll();dc.redrawAll();"`) provides the instructions on what to do when the 'reset' link is clicked on (in this case, we remove all the filters and redraw the dayOfWeekChart chart). Then there's a nice touch to *not* display the word reset when the page first loads (`style="display: none;"`) before finally printing the word 'reset' on the page.

The end result (when a day of the week is selected) looks like this;

Reset Link for the Row Chart

You can now click on the 'reset' link and the chart will revert to the default setting of all days selected.

Making the reset label a little bit better behaved.

While we now have our reset label working well, it's a bit poorly behaved the way that it creates a new line to put the label on. We can do better than that.

It would be fair to say that this is as a result of the decision to use the `<h4>` heading tags to make our chart headings. There are other options that could be employed to avoid using these, but I like them, so I'll describe how I kept them and kept the reset label on the same line.

None of what we're about to do is remotely d3.js or dc.js related. It's more HTML and CSS focussed (which doesn't mean it's not worth learning :-)).

The first thing we want to do is to get the 'reset' label onto the same line as our 'Day of the Week' heading.

This is simply done by ensuring that the <a> section is inside the <h4> section. The code should therefore look like this;

```
<div class='span4' id='dc-dayweek-chart'>
  <h4>Day of the Week
    <a class="reset"
      href="javascript:dayOfWeekChart.filterAll();dc.redrawAll();"
      style="display: none;">
      reset
    </a>
  </h4>
</div>
```

(Notice how the code layout shows the <a> code nested inside the <h4> section?)

The result on the web page now looks like this when a day is selected;

Reset Link for the Row Chart on the Same Line

That's a good start and certainly more acceptable, but the styling for the 'reset' label still looks a bit '**bold**' and '*BIG*'. We can do better than that.

What we'll do is place our <a> tag information inside a tag (this is the type of tag to use for in-line elements). Then we'll set a CSS style in our <stlye> area to make any text that is inside a which is inside a <h4> appear with formatting that makes it not bold and smaller in size.

First of all we place the <a> tag into a container like so;

```
<div class='span4' id='dc-dayweek-chart'>
  <h4>Day of the Week
    <span>
      <a class="reset"
        href="javascript:dayOfWeekChart.filterAll();dc.redrawAll();"
        style="display: none;">
        reset
      </a>
    </span>
  </h4>
</div>
```

Then we create a section at the start of our file (under the `<style type="text/css"></style>` line looks like the right place) that declares the styling for our h4 span text. It should look like this;

```
<style>
  h4 span {
    font-size:14px;
    font-weight:normal;
    }
</style>
```

That tells our web page that any `h4`, `span` labelled text should be `14px` in size and not bold (or `normal`).

The end result when you now have a day of the week selected looks like this;

Nicer Looking Reset Link for the Row Chart on the Same Line

Reset all the charts

We also have the option to reset all the charts at once. This could also be accomplished by reloading the page, but that would also incur a time and bandwidth penalty because the

associated data would be downloaded again. So just resetting everything in the browser is a good feature.

Again dc.js has got our back.

This feature is treated like a separate chart in itself, so it has a dimension and group and a section to draw the chart (not that it's a chart, but I'm sure you get the idea). It's executed slightly differently, but it's not too tricky.

What we're going to aim to do is provide our page with a title and add some nice dc.js trickery alongside that looks like this;

Reset All with Count Beside Page Title

The trickery shows us the number of selected records accompanied with the total number of records and gives us the option to reset all the selected charts so that all the records are selected.

There are 4 pieces of code that we will add to accomplish this task. We won't add them from top to bottom, because it makes slightly more sense to explain them in a different order.

First of all we will add the block of code that declares the variable that includes all of our data values (facts).

```
var all = facts.groupAll();
```

This piece of code should go soon after the line that initialises the crossfilter process (`var facts = crossfilter(data);`).

Then we will include a section of code that dimensions and counts all of our facts. It also anchors the values to the `dc-data-count` ID Selector that we will set up in a moment.

```
// count all the facts
dc.dataCount(".dc-data-count")
  .dimension(facts)
  .group(all);
```

This block of code belongs in the section that sets up our charts, although you could be forgiven for thinking that it kind of straddles more than one section.

The next section we'll add will be our title along with the count and reset information. It looks like this;

```
<div class="dc-data-count" style="float: left;">
  <h2>New Zealand Earthquakes
    <span>
      <span class="filter-count"></span>
       selected out of
      <span class="total-count"></span>
       records |
      <a href="javascript:dc.filterAll(); dc.renderAll();">Reset All</a>
    </span>
  </h2>
</div>
```

This block needs to go at the top of our area in the file where the layout of the portions of the web page are being set out. Put it directly under the outermost `container div` line (`<div class='container' style='font: 12px sans-serif;'>`).

It places a `<h2>` heading with the text 'New Zealand Earthquakes' and then places, in-line with this, five additional pieces. The first is a count of the filtered `facts` via...

```
<span class="filter-count"></span>
```

Then there is the text ' selected out of ' followed by a count of the total number of `facts` via...

```
<span class="total-count"></span>
```

The some more text ' records | ' and then another JavaScript call (as a link) that allows us to reset all the chart elements via...

```
<a href="javascript:dc.filterAll(); dc.renderAll();">Reset All</a>
```

This is all well and good, but the formatting will look a bit strange (like the following).

Reset All with Count Beside Page Title Poorly Formatted

This tells us that we need to apply some styling to the elements alongside the title. We can do this with the following CSS elements which can go into the `<style>` block with the one we added earlier for the other reset block.

```
h2 {
  float: right;
      }
h2 span {
  font-size:14px;
  font-weight:normal;
  }
```

These will allow the ‹h2› heading to be left justified and will reduce the size of the in-line span and remove the 'bold' formatting.

Et viola!

Nicer Looking Title with Count / Reset Information

Using Bootstrap with d3.js

Visualising data on a web page is a noble pursuit in itself, but often there is a need to be able to associate the visualization with other content (I know! It came as a surprise to me as well).

Developing a web page has become an activity that just about anyone can accomplish for better of for worse and I'm not going to claim to demonstrate any mastery of design or artistic flair. However, I have found using Bootstrap is a great way to make structural arrangements to a web page, it's simple to use and there is a fantastic range of features that can provide additional functionality to your pages and sometimes more importantly, a consistent 'feel' across *many* pages.

The syntax used for the Bootstrap code in this section is compatible with version 3.x. Previous versions of this book described the process using version 2.x so if there is some degree of confusion and you think is could be because you are using an older version of Bootstrap, I recommend that you move to version 3.x.

Bootstrap can be found here[371] and I thoroughly recommend working your way through the site to discover some of the cool things that it can do.

What is Bootstrap?

Twitter Bootstrap is a free collection of tools for creating websites and web applications. It contains HTML and CSS based design templates for typography, forms, buttons, charts, navigation and other interface components, as well as optional JavaScript extensions.

Bootstrap was developed by Mark Otto and Jacob Thornton at Twitter as a framework to encourage consistency across internal tools. The word 'framework' is probably the best descriptive term, since it's purpose is to provide structure to content. Perhaps in a similar way that d3.js provides structure to data.

Some of Bootstrap's most important features include;

- A layout grid
- Interface components

[371] http://getbootstrap.com/

Layout grid

Bootstrap includes 4 standard pixel width grid layout schemas which allow you to quickly arrange a page structure. This allows you to plan and implement what you're going to place on the page with a minimum of fuss. You can change any of the pre-set options if you wish and you can also implement a 'fluid' row option where bootstrap will dynamically size a column's width using a percentage instead of a fixed pixel value.

Bootstrap example page

It's this feature that first attracted me to using Bootstrap and while I may be using a complex tool for a simple task, it does that task very well.

Interface components

A large number of interface components are also provided. These include standard buttons...

Bootstrap Standard Buttons

... tables...

#	First Name	Last Name	Username
1	Mark	Otto	@mdo
2	Jacob	Thornton	@fat
3	Larry	the Bird	@twitter

#	First Name	Last Name	Username
1	Mark	Otto	@mdo
2	Jacob	Thornton	@fat
3	Larry	the Bird	@twitter

#	First Name	Last Name	Username
1	Mark	Otto	@mdo
	Mark	Otto	@TwBootstrap
2	Jacob	Thornton	@fat
3	Larry the Bird		@twitter

#	First Name	Last Name	Username
1	Mark	Otto	@mdo
2	Jacob	Thornton	@fat
3	Larry the Bird		@twitter

Bootstrap Tables

... labels...

Bootstrap Labels

... drop-down menus...

Bootstrap Drop-down menus

... navigation controls...

Bootstrap Navbars

... alerts...

Bootstrap Alerts

... and to be perfectly honest, the list goes on and on.

There is a dizzying array of options available for web designers and while I encourage you to use them, I can't promise to explain the nuances of their use, since I'm a humble journeyman in this world :-).

Incorporating Bootstrap into your html code.

Bootstrap is a remarkably flexible product. We could be forgiven for thinking that the process of installing it would be difficult. However, in the spirit of keeping things simple, we'll make the process crude, but effective.

You could easily just follow along with the instructions on the 'getting started[372]' page (and I recommend you do). But the following are important points.

Make sure you remember that you will need to download the appropriate scripts from the 'getting started[373]' page.;

Bootstrap Download

You will need to copy the `bootstrap.js` file (or the minimised version (`bootstrap.min.js`)) to a place where it can be reached and loaded by your script. While you're there, you will need to include a line to load the jquery.js file (which is a dependency of Bootstrap (not that it gets talked about much)) The following two lines, included with the line that loads d3.js, would do the job nicely (assuming that you've copied the `bootstrap.min.js` file into the `js` directory);

```
<script src="http://code.jquery.com/jquery.js"></script>
<script src="js/bootstrap.min.js"></script>
```

If we wanted to we could load Bootstrap the same way that we are loading jquery.js (off the Internet each time we load a page). To do this we could use;

```
<script src="http://code.jquery.com/jquery.js"></script>
<script src=
  "https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/js/bootstrap.min.js">
</script>
```

This is the way we will try to form our code in the coming examples to make the scripts as location independent as possible.

Make sure that the jquery line comes **before** the bootstrap line, because it won't work the other way round.

[372] http://getbootstrap.com/getting-started/
[373] http://twitter.github.io/bootstrap/getting-started.html

You will also need to copy the bootstrap.css (or the minimised version (`bootstrap.min.css`)) to a place where it can be reached and loaded by your script. The following lines show it being loaded from the css directory with the line that loads the script in the `<head>` section.

```
<head>

<link href="css/bootstrap.min.css" rel="stylesheet" media="screen">

</head>
```

Or again we could load it from the Internet as follows;

```
<head>

<link rel="stylesheet" href=
 "https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/css/bootstrap.min.css"
>

</head>
```

That should be all that's required! Of course as I mentioned earlier, there are plenty of other plug-in scripts that could be loaded to do fancy things with your web page, but we're going to try and keep things simple.

Arranging more than one graph on a web page.

We'll start with the presumption that we want to be able to display two separate graphs on the same web page. The example we will use is clearly contrived, but we should remember that it's the process we're interested in this case, not the content.

First make a page *with* two graphs

This is surprisingly easy. If you start with the simple graph that we initially used as our learning example at the start of the book, and duplicate the section that looks like the following, you are 99% of the way there.

```
// Adds the svg canvas
var        chart2 = d3.select("body")
        .append("svg")
                .attr("width", width + margin.left + margin.right)
                .attr("height", height + margin.top + margin.bottom)
        .append("g")
                .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

// Get the data
d3.csv("data2.csv", function(error, data) {
        data.forEach(function(d) {
                d.date = parseDate(d.date);
                d.close = +d.close;
        });

        // Scale the range of the data
        x.domain(d3.extent(data, function(d) { return d.date; }));
        y.domain([0, d3.max(data, function(d) { return d.close; })]);

        // Add the valueline path.
        chart2.append("path")
                .attr("class", "line")
                .attr("d", valueline(data));

        // Add the X Axis
        chart2.append("g")
                .attr("class", "x axis")
                .attr("transform", "translate(0," + height + ")")
                .call(xAxis);

        // Add the Y Axis
        chart2.append("g")
                .attr("class", "y axis")
```

```
                .call(yAxis);

});
```

For simplicity, the full code for this graph can also be found on github[374] or in the code samples bundled with this book (two-graphs-one-anchor.html, data-1.csv and data-2.csv). A live example can be found on bl.ocks.org[375].

The differences from the original simple graph example are;

- The graphs are slightly smaller (to make it easier to display the graphs as they move about).
- I have used *.csv files for the data and there are two different data files so that they look different and we can differentiate between the graphs.
- Most importantly, I have declared the two charts with different variable names (one as `chart1` and the other as `chart2`).

The different variable names are important, because if you leave them with the same identifier, the web page decides that what you're trying to do is to put all your drawing data into the same space. The end result is two graphs trying to occupy the same space and looks a bit like this...

Two Simple Graphs Mashed

The example with the correct (different) variable labels *should* look a little like this...

Two Simple Graphs

[374] https://gist.github.com/d3noob/5987480
[375] http://bl.ocks.org/d3noob/5987480

Arrange the graphs with the same anchor

The first thing I want to point out about how the graphs are presented is that they are both 'attached' to the same point on our web page. Both of the graphs select the body of the web page and then append a svg element to it;

```
var chart2 = d3.select("body")
    .append("svg")
```

This has the effect of appending the graphs to the *same* anchor point. Interestingly, if we narrow the window of our web browser to less that the width of both of our graphs side by side, the browser will automatically move one of the graphs to a position below the first in much the same way that text will wrap on a page.

Two Simple Graphs Wrapping

For a very simple mechanism of putting two graphs (or any two d3.js generated images) on a single page, this will work, but we don't have a lot of control over the positioning.

Arrange the graphs with separate anchors

To gain a little more control over where the graphs are placed we will employ ID selectors.

An ID selector is a way of naming an anchor point on an HTML page. They can be defined as "*a unique identifier to an element*". This means that we can name a position on our web page and then we can assign our graphs to those positions.

This can be done simply by placing `div` tags in our html file in an appropriate place (here I've put them in between the `<style>` section and the `<body>`).

```
</style>

<div id="area1"></div>
<div id="area2"></div>

<body>
```

Remembering that the tag defines a division or a section in an HTML document. Therefore we are labelling specific sections in our web page.

Now all we need to do is to tell each graph to append itself to either of these ID selectors. We do this by replacing the selected section in our JavaScript code with the appropriate ID selector as follows;

```
var chart1 = d3.select("#area1")
    .append("svg")
```

... and ...

```
var chart2 = d3.select("#area2")
    .append("svg")
```

A couple of points to note:

When we reference our ID selectors in the code (other than when we we set them with `id="area1"`) we need to put a hash (`#`) in front of the selector for the HTML to recognise it.

We can only use a single ID selector in a single place. This might sound like common sense, but whatever the temptation, don't go trying to assign the same ID selector to more than one place (You can certainly assign more than one item *to* an ID selector (for instance, you could append `chart2` to `area1` (`var chart2 = d3.select("#area1")`) but an ID selector is a ***unique*** identifier of a position.).

With these divs added, when you browse to the file, you will find that it looks like this;

Two Simple Graphs with divs

This looks the same as when the two graphs were wrapping when the browser was narrowed. However, this time the browser is wide enough to support the two side by side, but they won't position themselves that way. This is because each div *div*ides the web page. The top graph is in the div with the ID selector `area1` and the bottom graph is in the div with the ID selector `area2`. These divs effectively extend for the width of the web page.

The situation that we now find ourselves in is that we have control over where the graphs will be anchored, but we don't have much flexibility for arranging those anchors. This is where Bootstrap comes in.

How does Bootstrap's grid layout work

Bootstrap's grid layout subdivides the page by using rows and columns. A row will extend horizontally and a web page can be thought of as being 12 columns wide. Each column is a place to put horizontally separated content.

In order to make the best use of variable width devices, Bootstrap employs four different types of columns;

- Extra small devices (phones): With a horizontal resolution of less than 768px, Bootstrap reccomends we use `col-xs-`.
- Small devices (Tablets): With a horizontal resolution of greater than or equal to 768px, Bootstrap reccomends we use `col-sm-`.
- Medium devices (Desktops):With a horizontal resolution of greater than or equal to 992px, Bootstrap reccomends we use `col-md-`.
- Large devices (Desktops): With a horizontal resolution of greater than or equal to 1200px, Bootstrap reccomends we use `col-lg-`.

The examples we'll employ will use `col-md-`, but you should choose based on your anticipated needs.

Each column will have a width designated by a number at the end of our column designator. This will be the width in terms of the number of columns in the row.. For example `col-md-4` would be a single column 4 units wide (remember the row is a maximum of 12 possible units wide).

As an example, the picture below shows a single row divided into twelve individual columns.

Simple Bootstrap Layout with 12 columns

The columns can be combined to create larger spaces for larger content. The example below has a single `col-md-6` and two `col-md-3`'s.

Simple Bootstrap Layout with one column 6 and two column 3's

The column's will change height dynamically to fit their contents. So if there was a larger item in the `col-md-6` example given above (perhaps a graph), it would expand like so;

Simple Bootstrap Layout with Content

The way to set these rows and columns up is by *div*iding the screen using divs and assigning them class types that match the grid layout.

For example, to create our example of a single row with a `col-md-6` and two `col-md-3`'s we would use the following html code as our baseline.

```
<div class="row">
  <div class="col-md-6"></div>
  <div class="col-md-3"></div>
  <div class="col-md-3"></div>
</div>
```

In this example code we can see the `row` div is enclosing the three columns. We can extend the comparison by putting the code into our graphic example.

Simple Bootstrap Layout with Content and Code

To add content to the structure, all that is needed is to put our web page components between the `<div class="col-md-x">` and `</div>` tags.

Later we will look (briefly) at more complex configurations that might be useful.

Arrange more than one d3.js graph with Bootstrap

In the previous sections we have seen how to assign ID selectors so that we anchor our d3.js graphs to a particular section of our web page. We have also seen how to utilise Bootstrap to divide up our web page into different sections. Now we will bring the two examples together and assign ID selectors to sections set up with Bootstrap.

We will start with our simple two graph example (as seen on bl.ocks.org here[376]).

We will need to make sure we have our `bootstrap.min.js` and `bootstrap.min.css` files in the appropriate place.

Then insert the code to use `bootstrap.min.css` at the start of the file (just before the `<style>` tag would be good);

```
<head>

<link rel="stylesheet" href=
 "https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/css/bootstrap.min.css"
>

</head>
```

Then include the lines to load the `jquery.js` and `bootstrap.min.js` files just after the line that loads the `d3.js` file.

```
<script src="http://code.jquery.com/jquery.js"></script>
<script src=
  "https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/js/bootstrap.min.js">
</script>
```

What we'll do to make things simple is to create a Bootstrap layout that is made up of a single row with just two `col-md-6` elements in it. The following code will do this nicely and should go after the `</style>` tag and before the `<body>` tag.

```
<div class="row">
  <div class="col-md-6"></div>
  <div class="col-md-6"></div>
</div>
```

Now we add in our ID selectors in a clever way by incorporating them into the divs that we have just entered. So remembering the code for our original two selectors...

[376] http://bl.ocks.org/d3noob/raw/5987480/

```
<div id="area1"></div>
<div id="area2"></div>
```

... we can incorporate these into our row and columns as follows;

```
<div class="row">
  <div class="col-md-6" id="area1"></div>
  <div class="col-md-6" id="area2"></div>
</div>
```

The last thing we need to do is to change the `d3.select` from selecting the `body` of the web page to selecting our two new ID selectors `area1` and `area2`.

```
var chart1 = d3.select("#area1")
    .append("svg")
```

... and ...

```
var chart2 = d3.select("#area2")
    .append("svg")
```

Et viola! Our new web page has two graphs which are settled into their own specific section.

Simple Bootstrap Layout Example with Graphs

To provide another example of the flexibility of the layout schema, we can take our row / column layout section and adapt it so that our graphs are in two separate sections with a third, smaller, section in the middle describing the graphs.

If we start with our previously entered columns with their ID selectors;

```
<div class="row">
  <div class="col-md-6" id="area1"></div>
  <div class="col-md-6" id="area2"></div>
</div>
```

We can change the columns to `col-md-5` and add an additional `col-md-2` in between with some text (remember, the total number of columns has to add up to 12).

```
<div class="row">
  <div class="col-md-5" id="area1"></div>
  <div class="col-md-2">
    To the left is a graph showing the anticipated profits
    of the 'Widget Incorporated' company.
    On the right is the anticipated cost of production as
    the number of Widgets is increased.
    Clearly we will be RICH!
  </div>
  <div class="col-md-5" id="area2"></div>
</div>
```

And the end result is...

Simple Bootstrap Layout with Graphs and Text

Neither of these examples is particularly elegant in terms of its layout. I am relying on you to bring the prettiness!

The full code for this final graph and paragraph combination can also be found on github[377] or in the code samples bundled with this book (two-graphs-bootstrap.html, data-1.csv and data-2.csv). A live example can be found on bl.ocks.org[378].

A more complicated Bootstrap layout example

As promised earlier, it's worth looking at a more complex example for a layout with Bootstrap, just to get a feel for how it works or the potential it might have for you.

[377] https://gist.github.com/d3noob/6a3b59149cf3ebdb3fc4
[378] http://bl.ocks.org/d3noob/6a3b59149cf3ebdb3fc4

The example code layout we will design will look a bit like this;

More Complicated Bootstrap Layout

It looks slightly complex with a nesting of columns and rows, and the end result is only 5 separate sections, but it's really not too hard to put together if you start in the right place and build it up piece by piece.

We'll start in the middle and work our way out. The first piece to consider is the two side-by-side `col-md-4`'s.

Two col-md-4's

The code for these is just...

```
<div class="row">
  <div class="col-md-4"></div>
  <div class="col-md-4"></div>
</div>
```

Directly under that row is another with a single `col-md-8`.

A Single col-md-8

The code for this section is...

```
<div class="row">
  <div class="col-md-8">
</div>
```

Both of these rows together look like this;

Two Stacked rows

And the code is just one piece after the other.

```
<div class="row">
  <div class="col-md-n4"></div>
  <div class="col-md-4"></div>
</div>
<div class="row">
  <div class="col-md-8"></div>
</div>
```

Because this entire block forms part of another (larger) row, we need to enclose it in its own `col-md-8` (since this is part is only `col-md-8` wide).

Enclosed Stacked rows

And for the code the new `col-md-8` div wraps all the current code we have.

```
<div class="col-md-8">
  <div class="row">
    <div class="col-md-4"></div>
    <div class="col-md-4"></div>
  </div>
  <div class="row">
    <div class="col-md-8"></div>
  </div>
</div>
```

The `col-md-8` is alongside a large `col-md-4` that sits to the left.

col-md-4 plus Complex col-md-8

This requires another `col-md-4` div to be placed before the `col-md-8`.

```
<div class="col-md-4"></div>
<div class="col-md-8">
  <div class="row">
    <div class="col-md-4"></div>
    <div class="col-md-4"></div>
  </div>
  <div class="row">
    <div class="col-md-8"></div>
  </div>
</div>
```

The `col-md-4` and the complex `col-md-8` need to be in their own row...

col-md-4 plus Complex col-md-8 in a `row`

So a `row` div encloses all the code we have so far.

```
<div class="row">
  <div class="col-md-4"></div>
  <div class="col-md-8">
    <div class="row">
      <div class="col-md-4"></div>
      <div class="col-md-4"></div>
    </div>
    <div class="row">
      <div class="col-md-8"></div>
    </div>
  </div>
</div>
```

Finally we need to place another row with a `col-md-12` in it above our current work.

More Complicated Bootstrap Layout

Again, we need to place the row and column before our current code so that it appears *above* the current code on the page.

```
<div class="row">
  <div class="col-md-12"></div>
</div>
<div class="row">
  <div class="col-md-4"></div>
  <div class="col-md-8">
    <div class="row">
      <div class="col-md-4"></div>
      <div class="col-md-4"></div>
    </div>
    <div class="row">
      <div class="col-md-8"></div>
    </div>
  </div>
</div>
```

There we have it!

Slightly more complex, but if you needed a heading, a sidebar, a couple of graphs and some explanatory text, that might be exactly what you were looking for :-).

Working with GitHub, Gist and bl.ocks.org

General stuff about bl.ocks.org

In the words of Mike Bostock on the bl.ocks.org[379] main page;

> *"This is a simple viewer for code examples hosted on GitHub Gist. Code up an example using Gist, and then point people here to view the example and the source code, live!"*

The whole idea is to take the information that you have in a gist (the pastebin area in Github) and to give it a viewer that will allow it to display in your browser.

The reason this works is that the files that make up a web page that can be displayed in your browser conform to a pretty well defined standard. If you can name your main web file index.html and put it in a gist, bl.ocks.org will not just render it to a browser, but since you can store your data files in the same gists, your visualization can use those as data sources as well since they shouldn't violate any cross domain security restrictions.

Mike's clever code allows a gallery type preview page to be generated (including a thumbnails if you follow the instructions in another part of this section).

Thumbnails of examples for d3noob's blocks

And if you include a readme file formatted using markdown you can have a nice little explanation of how your visualization works.

The front rendering page includes any markdown notes and the code (not the full screen) is optimised to accept visualizations of 960x500 pixels (although you can make them other sizes, it's just that this is an 'optimum' size). Of course there is always the full screen mode to render your creation in its full glory if necessary.

If I was to pass on any advice when using bl.ocks.org, please consider others who will no doubt view your work and wonder how you achieved your magic. Help them along where possible with a few comments in the readme.md file because sharing is caring :-).

[379] http://bl.ocks.org/

Installing the plug-in for bl.ocks.org for easy block viewing

This might sound slightly odd at first if you're not familiar with using Gist or bl.ocks.org, but trust me, a) you should use them, b) if you get to the point where you are using these fantastic services, there's a good chance that you will want to be able to quickly check out what your block looks like when you update or add in a Gist.

Here's the scenario. You're slaving away getting all your data and files into Gist, and then you're switching - in some tiresome manner - to get to the block that bl.ocks.org generates.

Well, throw away that tiresome technique! It's time to move into the 21st century with some plug-in goodness. Clever Mike Bostock has put together some handy dandy browser extensions that will add a button to your Chrome, Safari or Firefox browser to take you straight from your Gist to your block!

It will turn your Gist page from this...

Gist page without bl.ocks.org button

... to this ...

Gist page with bl.ocks.org button!

Check out the button!

It's really handy and works like a charm. You can download it directly from the bl.ocks.org home page[380] or from theGithub page[381] where the code is hosted (this also includes a quick couple of lines of instructions for installation if you're unsure).

Loading a thumbnail into Gist for bl.ocks.org d3 graphs

This description will start on the assumption that the user already has a GitHub / Gist account set up and running. It's purpose is to demonstrate how to upload an image as a file named thumbnail.png to a Gist so that when viewing the users home page on bl.ocks.org you see a nice little preview of what a visitor can anticipate, when they go to look at your work :-). This description is a fleshed out version of the one provided by Christophe Viau on Google Groups[382].

Setting the scene:

There you are: a fresh faced d3.js user keen to share his/her work with the world. You set yourself up a GitHub / Gist account and put your code into a gist.

[380] http://bl.ocks.org/
[381] https://github.com/mbostock/bl.ocks.org
[382] https://groups.google.com/forum/?fromgroups=#!topic/d3-js/FBosXiTB9Pc

The gist web page

Your graph is a thing of rare beauty and the community needs to marvel at your brilliance. Of course this is a breeze with bl.ocks.org. Once you have all the code sorted out, and all data files made accessible, bl.ocks.org can display the graph with the code and can even open the graph in its own window. The person responsible for bl.ocks.org? Mike Bostock of course (wherever does he get the time?).

Clicking on the bl.ocks.org button on the gist page (load the extension available from the main page of bl.ocks.org) takes you to see your graph.

Your awesome graph ready to go

Wow! Impressive.

So you think that will make a fine addition to your collection of awesome graphs and if you click on your GitHub user name that is in the top left of the screen you go to a page that lays out all your graphs with a thumbnail giving a sneak preview of what the user can expect.

d3noob's blocks, but no thumbnail!

Aww... Rats! There's a nice place holder, but no pretty picture.

Hang on, what had Mike said on the bl.ocks.org main page?

"The main source code for your example should be named index.html. You can also include a README.md using Markdown, and a thumbnail.png for preview."

Ahh.. you need to include a thumbnail.png file in your Gist!

So how to get it there? Well Gist is a repository, so what you need to do is to put the code in there somehow. Now from the Gist web page this doesn't appear to be a nice (gui) way to do this. So from here you will need to suspend your noob status and hit the command line.

The good news (if you're a windows user (and sorry, I haven't done this in Linux or on a Mac)) is that, as part of the GitHub for windows installation, a command line tool was installed as well! Prepare yourself, you're going to use the Git Shell.

The Windows GitHub and Git Shell icons

Enough of the scene setting. Let's git going :-).

I'm going to describe the steps in a pretty verbose fashion with pretty pictures and everything else, but at the end I will put a simple set of steps in the form that Christophe Viau outlined on Google Groups[383].

First you will want to have your image ready. It needs to be a png with dimensions of 230 x 120 pixels. It should also be less than 50kB in size.

Go to your public Gist that you have already set up and copy the link in the "Clone this gist" box.

Copy the 'Clone this gist' link

(this should look something like https://gist.github.com/441443[384])

Now you're going to clone this gist to a local repository using the Git Shell. Open it up from the desktop icon and you should see something like the following;

The Git Shell is open for business

You can clone the gist to a local folder with the command;

```
git clone https://gist.github.com/4414436.git
```

Or if you're using OSX, the following command has been passed on by Alex Hornbake as an alternative (thanks Alex).

```
git clone git@gist.github.com:4414436.git
```

[383] https://groups.google.com/forum/?fromgroups=#!topic/d3-js/FBosXiTB9Pc
[384] https://gist.github.com/441443

(The url is the one copied from the 'Clone this gist' box.)

Running the command

This will create a folder with the id (the number) of the gist in your local GitHub working directory.

A folder is created for your gist

And there it is (Ooo... Look almost New Years!).

Another tip for Mac users (This time from 'Fern of the Andes' in a comment from the d3noob.org[385] blog) is that at this stage you should be able to drag the generated folder into GitHub for Mac (http://mac.github.com/). Then simply drag the image file into the GitHub for Mac folder and commit. Have a crack if you're confident enough. I can't claim to have tried it, but Fern reckons it's OK :-).

Copy your thumbnail.png file into this directory.

Back to the Git Shell and change into the directory (4414436) . We can now add the thumbnail.png file to the gist with the command;

```
git add thumbnail.png
```

Running the git add command

And now commit it to your gist with the following command in the Git Shell;

```
git commit -m "Thumbnail image added"
```

Running the git commit command

Now we need to push the commit to the remote gist (you may be asked for your GitHub user name and password if you haven't done this before) with the following command;

[385] http://www.d3noob.org/2012/12/loading-thumbnail-into-gist-for.html

```
git push
```

Push! Push!

OK, now you can go back to the web page for your gist and refresh it and scroll on down…

A thumbnail is born

Woo Hoo!

(I know it doesn't look like much, but this is a VERY simple graph :-)).

Now for the real test. Go back to your home page for your blocks on bl.ocks.org and refresh the page.

d3noob's blocks complete with thumbnail

Oh yes. You may now bask in the sweet glow of victory. And as a little bit of extra fancy, if you move your mouse over the image it translates up slightly!

Wrap up.

The steps to get your thumbnail into the gist aren't exactly point and click, but the steps you need to take are fairly easy to follow. As promised, here is the abridged list of steps that will avoid you going through the several previous pages.

1. Create your public gist on https://gist.github.com/[386]
2. Get an image ready (230 x 120 pixels, named thumbnail.png)
3. Under "Clone this gist", copy the link (i.e., https://gist.github.com/4414436.git)
4. If you have the command line git tools (Git Shell), clone this gist to a local folder: `git clone https://gist.github.com/4414436.git` (or `git clone git@gist.github.com:4414436.git` for OSX) It will add a folder with the gist id as a name (i.e., 4414436) under the current working directory.

[386] https://gist.github.com/

5. Navigate to this folder via the command line in Git Shell: `cd 4414436` (dir `4414436` on windows)
6. Navigate to this folder in file explorer and add your image (i.e., thumbnail.png)
7. Add it to git from the command line: `git add thumbnail.png`
8. Commit it to git: `git commit -m "Thumbnail added"`
9. Push this commit to your remote gist (you may need your Github user name and password): `git push`
10. Go back and refresh your Gist on https://gist.github.com/ to confirm that it worked
11. Check your blocks home page and see if it's there too. http://bl.ocks.org/<*yourusername*>

Just to finish off. A big thanks to Christophe Viau for the hard work on finding out how it all goes together and if there are any errors in the above description I have no doubt they will be mine.

Appendices

Simple Line Graph

```
<!DOCTYPE html>
<meta charset="utf-8">
<style> /* set the CSS */

body { font: 12px Arial;}

path {
    stroke: steelblue;
    stroke-width: 2;
    fill: none;
}

.axis path,
.axis line {
    fill: none;
    stroke: grey;
    stroke-width: 1;
    shape-rendering: crispEdges;
}

</style>
<body>

<!-- load the d3.js library -->
<script src="http://d3js.org/d3.v3.min.js"></script>

<script>

// Set the dimensions of the canvas / graph
var margin = {top: 30, right: 20, bottom: 30, left: 50},
    width = 600 - margin.left - margin.right,
    height = 270 - margin.top - margin.bottom;

// Parse the date / time
var parseDate = d3.time.format("%d-%b-%y").parse;

// Set the ranges
var x = d3.time.scale().range([0, width]);
```

```
var y = d3.scale.linear().range([height, 0]);

// Define the axes
var xAxis = d3.svg.axis().scale(x)
    .orient("bottom").ticks(5);

var yAxis = d3.svg.axis().scale(y)
    .orient("left").ticks(5);

// Define the line
var valueline = d3.svg.line()
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y(d.close); });

// Adds the svg canvas
var svg = d3.select("body")
    .append("svg")
        .attr("width", width + margin.left + margin.right)
        .attr("height", height + margin.top + margin.bottom)
    .append("g")
        .attr("transform",
              "translate(" + margin.left + "," + margin.top + ")");

// Get the data
d3.csv("data/data.csv", function(error, data) {
    data.forEach(function(d) {
        d.date = parseDate(d.date);
        d.close = +d.close;
    });

    // Scale the range of the data
    x.domain(d3.extent(data, function(d) { return d.date; }));
    y.domain([0, d3.max(data, function(d) { return d.close; })]);

    // Add the valueline path.
    svg.append("path")
        .attr("class", "line")
        .attr("d", valueline(data));

    // Add the X Axis
    svg.append("g")
        .attr("class", "x axis")
        .attr("transform", "translate(0," + height + ")")
        .call(xAxis);

    // Add the Y Axis
```

```
    svg.append("g")
        .attr("class", "y axis")
        .call(yAxis);

});

</script>
</body>
```

Graph with Many Features

```
<!DOCTYPE html>
<meta charset="utf-8">
<style>

body {
    font: 12px Arial;
}

text.shadow {
  stroke: #fff;
  stroke-width: 2.5px;
  opacity: 0.9;
}

path {
    stroke: steelblue;
    stroke-width: 2;
    fill: none;
}

.axis path,
.axis line {
    fill: none;
    stroke: grey;
    stroke-width: 1;
    shape-rendering: crispEdges;
}

.grid .tick {
    stroke: lightgrey;
    stroke-opacity: 0.7;
    shape-rendering: crispEdges;
}
.grid path {
          stroke-width: 0;
}

.area {
    fill: lightsteelblue;
      stroke-width: 0;
}

</style>
```

```
<body>
<script src="http://d3js.org/d3.v3.min.js"></script>

<script>

var margin = {top: 30, right: 20, bottom: 35, left: 50},
    width = 600 - margin.left - margin.right,
    height = 270 - margin.top - margin.bottom;

var parseDate = d3.time.format("%d-%b-%y").parse;

var x = d3.time.scale().range([0, width]);
var y = d3.scale.linear().range([height, 0]);

var xAxis = d3.svg.axis()
    .scale(x)
    .orient("bottom")
    .ticks(5);

var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left")
    .ticks(5);

var area = d3.svg.area()
    .x(function(d) { return x(d.date); })
    .y0(height)
    .y1(function(d) { return y(d.close); });

var valueline = d3.svg.line()
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y(d.close); });

var svg = d3.select("body")
    .append("svg")
        .attr("width", width + margin.left + margin.right)
        .attr("height", height + margin.top + margin.bottom)
    .append("g")
        .attr("transform",
              "translate(" + margin.left + "," + margin.top + ")");

// function for the x grid lines
function make_x_axis() {
    return d3.svg.axis()
        .scale(x)
        .orient("bottom")
```

```
        .ticks(5)
}

// function for the y grid lines
function make_y_axis() {
  return d3.svg.axis()
      .scale(y)
      .orient("left")
      .ticks(5)
}

// Get the data
d3.csv("data.csv", function(error, data) {
    data.forEach(function(d) {
        d.date = parseDate(d.date);
        d.close = +d.close;
    });

    // Scale the range of the data
    x.domain(d3.extent(data, function(d) { return d.date; }));
    y.domain([0, d3.max(data, function(d) { return d.close; })]);

    // Add the filled area
    svg.append("path")
        .datum(data)
        .attr("class", "area")
        .attr("d", area);

    // Draw the x Grid lines
    svg.append("g")
        .attr("class", "grid")
        .attr("transform", "translate(0," + height + ")")
        .call(make_x_axis()
            .tickSize(-height, 0, 0)
            .tickFormat("")
        )

    // Draw the y Grid lines
    svg.append("g")
        .attr("class", "grid")
        .call(make_y_axis()
            .tickSize(-width, 0, 0)
            .tickFormat("")
        )

    // Add the valueline path.
```

```
svg.append("path")
    .attr("d", valueline(data));

// Add the X Axis
svg.append("g")
    .attr("class", "x axis")
    .attr("transform", "translate(0," + height + ")")
    .call(xAxis);

// Add the Y Axis
svg.append("g")
    .attr("class", "y axis")
    .call(yAxis);

// Add the text label for the X axis
svg.append("text")
    .attr("transform",
          "translate(" + (width/2) + " ," +
                         (height+margin.bottom) + ")")
    .style("text-anchor", "middle")
    .text("Date");

// Add the white background to the y axis label for legibility
svg.append("text")
    .attr("transform", "rotate(-90)")
    .attr("y", 6)
    .attr("x", margin.top - (height / 2))
    .attr("dy", ".71em")
    .style("text-anchor", "end")
    .attr("class", "shadow")
    .text("Price ($)");

// Add the text label for the Y axis
svg.append("text")
    .attr("transform", "rotate(-90)")
    .attr("y", 6)
    .attr("x", margin.top - (height / 2))
    .attr("dy", ".71em")
    .style("text-anchor", "end")
    .text("Price ($)");

// Add the title
svg.append("text")
    .attr("x", (width / 2))
    .attr("y", 0 - (margin.top / 2))
    .attr("text-anchor", "middle")
```

```
        .style("font-size", "16px")
        .style("text-decoration", "underline")
        .text("Price vs Date Graph");

});

</script>
</body>
```

Graph with Area Gradient

```
<!DOCTYPE html>
<meta charset="utf-8">
<style>


body { font: 12px Arial;}
.axis path,
.axis line {
    fill: none;
    stroke: grey;
    stroke-width: 1;
    shape-rendering: crispEdges;
}
.area {                        /* changed from line to area */
  fill: url(#area-gradient);  /* url reference fill instead of stroke */
  stroke-width: 0px;          /* removed stroke reference and any line*/
}


</style>
<body>
<script type="text/javascript" src="d3/d3.v3.js"></script>
<script>


// Set the dimensions of the canvas / graph
var margin = {top: 30, right: 20, bottom: 30, left: 50},
    width = 600 - margin.left - margin.right,
    height = 270 - margin.top - margin.bottom;
// Parse the date / time
var parseDate = d3.time.format("%d-%b-%y").parse;
// Set the ranges
var x = d3.time.scale().range([0, width]);
var y = d3.scale.linear().range([height, 0]);
// Define the axes
var xAxis = d3.svg.axis().scale(x)
    .orient("bottom").ticks(5);
var yAxis = d3.svg.axis().scale(y)
    .orient("left").ticks(5);
// Define the area (remove the line definition)
var area = d3.svg.area()
    .x(function(d) { return x(d.date); })
    .y0(height)
    .y1(function(d) { return y(d.close); });
```

```
// Adds the svg canvas
var svg = d3.select("body")
    .append("svg")
        .attr("width", width + margin.left + margin.right)
        .attr("height", height + margin.top + margin.bottom)
    .append("g")
        .attr("transform",
              "translate(" + margin.left + "," + margin.top + ")"
             );
// Get the data
d3.tsv("data/data.tsv", function(error, data) {
    data.forEach(function(d) {
        d.date = parseDate(d.date);
        d.close = +d.close;
    });
    // Scale the range of the data
    x.domain(d3.extent(data, function(d) { return d.date; }));
    y.domain([0, d3.max(data, function(d) { return d.close; })]);
    // Set the threshold
    svg.append("linearGradient")
        .attr("id", "area-gradient")      // change from line to area
        .attr("gradientUnits", "userSpaceOnUse")
        .attr("x1", 0).attr("y1", y(0))
        .attr("x2", 0).attr("y2", y(1000))
    .selectAll("stop")
        .data([
            {offset: "0%", color: "red"},
            {offset: "30%", color: "red"},
            {offset: "45%", color: "black"},
            {offset: "55%", color: "black"},
            {offset: "60%", color: "lawngreen"},
            {offset: "100%", color: "lawngreen"}
        ])
    .enter().append("stop")
        .attr("offset", function(d) { return d.offset; })
        .attr("stop-color", function(d) { return d.color; });
    // Add the filled area and remove the value line block
    svg.append("path")
        .datum(data)
        .attr("class", "area")
        .attr("d", area);
    // Add the X Axis
    svg.append("g")
        .attr("class", "x axis")
        .attr("transform", "translate(0," + height + ")")
        .call(xAxis);
```

```
    // Add the Y Axis
    svg.append("g")
        .attr("class", "y axis")
        .call(yAxis);
});


</script>
</body>
```

Bar Chart

```
<!DOCTYPE html>
<meta charset="utf-8">

<head>
        <style>

        .axis {
          font: 10px sans-serif;
        }

        .axis path,
        .axis line {
          fill: none;
          stroke: #000;
          shape-rendering: crispEdges;
        }

        </style>
</head>

<body>

<script src="http://d3js.org/d3.v3.min.js"></script>

<script>

var margin = {top: 20, right: 20, bottom: 70, left: 40},
    width = 600 - margin.left - margin.right,
    height = 300 - margin.top - margin.bottom;

// Parse the date / time
var        parseDate = d3.time.format("%Y-%m").parse;

var x = d3.scale.ordinal().rangeRoundBands([0, width], .05);

var y = d3.scale.linear().range([height, 0]);

var xAxis = d3.svg.axis()
    .scale(x)
    .orient("bottom")
    .tickFormat(d3.time.format("%Y-%m"));

var yAxis = d3.svg.axis()
```

```
    .scale(y)
    .orient("left")
    .ticks(10);

var svg = d3.select("body").append("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
  .append("g")
    .attr("transform",
          "translate(" + margin.left + "," + margin.top + ")");

d3.csv("bar-data.csv", function(error, data) {

    data.forEach(function(d) {
        d.date = parseDate(d.date);
        d.value = +d.value;
    });

  x.domain(data.map(function(d) { return d.date; }));
  y.domain([0, d3.max(data, function(d) { return d.value; })]);

  svg.append("g")
      .attr("class", "x axis")
      .attr("transform", "translate(0," + height + ")")
      .call(xAxis)
    .selectAll("text")
      .style("text-anchor", "end")
      .attr("dx", "-.8em")
      .attr("dy", "-.55em")
      .attr("transform", "rotate(-90)" );

  svg.append("g")
      .attr("class", "y axis")
      .call(yAxis)
    .append("text")
      .attr("transform", "rotate(-90)")
      .attr("y", 6)
      .attr("dy", ".71em")
      .style("text-anchor", "end")
      .text("Value ($)");

  svg.selectAll("bar")
      .data(data)
    .enter().append("rect")
      .style("fill", "steelblue")
      .attr("x", function(d) { return x(d.date); })
```

```
      .attr("width", x.rangeBand())
      .attr("y", function(d) { return y(d.value); })
      .attr("height", function(d) { return height - y(d.value); });

});

</script>

</body>
```

Linking Objects

```
<!DOCTYPE html>
<meta charset="utf-8">

<body>

<!-- load the d3.js library -->
<script src="http://d3js.org/d3.v3.min.js"></script>

<script>

var width = 449;
var height = 249;
var word = "gongoozler";

var holder = d3.select("body")
      .append("svg")
      .attr("width", width)
      .attr("height", height);

// draw a rectangle
holder.append("a")
    .attr("xlink:href", "http://en.wikipedia.org/wiki/"+word)
    .append("rect")
    .attr("x", 100)
    .attr("y", 50)
    .attr("height", 100)
    .attr("width", 200)
    .style("fill", "lightgreen")
    .attr("rx", 10)
    .attr("ry", 10);

// draw text on the screen
holder.append("text")
    .attr("x", 200)
    .attr("y", 100)
    .style("fill", "black")
    .style("font-size", "20px")
    .attr("dy", ".35em")
    .attr("text-anchor", "middle")
    .style("pointer-events", "none")
    .text(word);

</script>
```

```
</body>
```

A live version of this code can be found online on bl.ocks.org[387] and GitHub[388].

[387] http://bl.ocks.org/d3noob/8150631
[388] https://gist.github.com/d3noob/8150631

PHP with MySQL Access

```
<?php
    $username = "homedbuser";
    $password = "homedbuser";
    $host = "localhost";
    $database="homedb";

    $server = mysql_connect($host, $username, $password);
    $connection = mysql_select_db($database, $server);

    $myquery = "
SELECT  `date`, `close` FROM  `data2`
";
    $query = mysql_query($myquery);

    if ( ! $myquery ) {
        echo mysql_error();
        die;
    }

    $data = array();

    for ($x = 0; $x < mysql_num_rows($query); $x++) {
        $data[] = mysql_fetch_assoc($query);
    }

    echo json_encode($data);

    mysql_close($server);
?>
```

Simple Sankey Graph

```
<!DOCTYPE html>
<meta charset="utf-8">
<title>SANKEY Experiment</title>
<style>


.node rect {
  cursor: move;
  fill-opacity: .9;
  shape-rendering: crispEdges;
}
.node text {
  pointer-events: none;
  text-shadow: 0 1px 0 #fff;
}
.link {
  fill: none;
  stroke: #000;
  stroke-opacity: .2;
}
.link:hover {
  stroke-opacity: .5;
}


</style>
<body>
<p id="chart">
<script type="text/javascript" src="d3/d3.v3.js"></script>
<script src="js/sankey.js"></script>
<script>


var units = "Widgets";
var margin = {top: 10, right: 10, bottom: 10, left: 10},
    width = 700 - margin.left - margin.right,
    height = 300 - margin.top - margin.bottom;
var formatNumber = d3.format(",.0f"),    // zero decimal places
    format = function(d) { return formatNumber(d) + " " + units; },
    color = d3.scale.category20();
// append the svg canvas to the page
var svg = d3.select("#chart").append("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
```

```
  .append("g")
    .attr("transform",
          "translate(" + margin.left + "," + margin.top + ")");
// Set the sankey diagram properties
var sankey = d3.sankey()
    .nodeWidth(36)
    .nodePadding(40)
    .size([width, height]);
var path = sankey.link();
// load the data
d3.json("data/sankey-formatted.json", function(error, graph) {
  sankey
      .nodes(graph.nodes)
      .links(graph.links)
      .layout(32);
// add in the links
  var link = svg.append("g").selectAll(".link")
      .data(graph.links)
    .enter().append("path")
      .attr("class", "link")
      .attr("d", path)
      .style("stroke-width", function(d) { return Math.max(1, d.dy); })
      .sort(function(a, b) { return b.dy - a.dy; });
// add the link titles
  link.append("title")
        .text(function(d) {
                    return d.source.name + " □ " +
                d.target.name + "\n" + format(d.value); });
// add in the nodes
  var node = svg.append("g").selectAll(".node")
      .data(graph.nodes)
    .enter().append("g")
      .attr("class", "node")
      .attr("transform", function(d) {
                  return "translate(" + d.x + "," + d.y + ")"; })
    .call(d3.behavior.drag()
      .origin(function(d) { return d; })
      .on("dragstart", function() {
                  this.parentNode.appendChild(this); })
      .on("drag", dragmove));
// add the rectangles for the nodes
  node.append("rect")
      .attr("height", function(d) { return d.dy; })
      .attr("width", sankey.nodeWidth())
      .style("fill", function(d) {
                  return d.color = color(d.name.replace(/ .*/, "")); })
```

```
      .style("stroke", function(d) {
                  return d3.rgb(d.color).darker(2); })
    .append("title")
      .text(function(d) {
                  return d.name + "\n" + format(d.value); });
// add in the title for the nodes
  node.append("text")
      .attr("x", -6)
      .attr("y", function(d) { return d.dy / 2; })
      .attr("dy", ".35em")
      .attr("text-anchor", "end")
      .attr("transform", null)
      .text(function(d) { return d.name; })
    .filter(function(d) { return d.x < width / 2; })
      .attr("x", 6 + sankey.nodeWidth())
      .attr("text-anchor", "start");
// the function for moving the nodes
  function dragmove(d) {
    d3.select(this).attr("transform",
        "translate(" + (
            d.x = Math.max(0, Math.min(width - d.dx, d3.event.x))
        )
        + "," + (
            d.y = Math.max(0, Math.min(height - d.dy, d3.event.y))
        ) + ")");
    sankey.relayout();
    link.attr("d", path);
  }
});


</script>
</body>
</html>
```

Simple Tree Diagram

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">

    <title>Collapsible Tree Example</title>

    <style>

        .node circle {
          fill: #fff;
          stroke: steelblue;
          stroke-width: 3px;
        }

        .node text { font: 12px sans-serif; }

        .link {
          fill: none;
          stroke: #ccc;
          stroke-width: 2px;
        }

    </style>

  </head>

  <body>

<!-- load the d3.js library -->
<script src="http://d3js.org/d3.v3.min.js"></script>

<script>

var treeData = [
  {
    "name": "Top Level",
    "parent": "null",
    "children": [
      {
        "name": "Level 2: A",
        "parent": "Top Level",
        "children": [
```

```
          {
            "name": "Son of A",
            "parent": "Level 2: A"
          },
          {
            "name": "Daughter of A",
            "parent": "Level 2: A"
          }
        ]
      },
      {
        "name": "Level 2: B",
        "parent": "Top Level"
      }
    ]
  }
];

// ************** Generate the tree diagram         *****************
var margin = {top: 20, right: 120, bottom: 20, left: 120},
        width = 960 - margin.right - margin.left,
        height = 500 - margin.top - margin.bottom;

var i = 0;

var tree = d3.layout.tree()
        .size([height, width]);

var diagonal = d3.svg.diagonal()
        .projection(function(d) { return [d.y, d.x]; });

var svg = d3.select("body").append("svg")
        .attr("width", width + margin.right + margin.left)
        .attr("height", height + margin.top + margin.bottom)
  .append("g")
        .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

root = treeData[0];

update(root);

function update(source) {

  // Compute the new tree layout.
  var nodes = tree.nodes(root).reverse(),
          links = tree.links(nodes);
```

```
  // Normalize for fixed-depth.
  nodes.forEach(function(d) { d.y = d.depth * 180; });

  // Declare the nodes…
  var node = svg.selectAll("g.node")
          .data(nodes, function(d) { return d.id || (d.id = ++i); });

  // Enter the nodes.
  var nodeEnter = node.enter().append("g")
          .attr("class", "node")
          .attr("transform", function(d) {
                  return "translate(" + d.y + "," + d.x + ")"; });

  nodeEnter.append("circle")
          .attr("r", 10)
          .style("fill", "#fff");

  nodeEnter.append("text")
          .attr("x", function(d) {
                  return d.children || d._children ? -13 : 13; })
          .attr("dy", ".35em")
          .attr("text-anchor", function(d) {
                  return d.children || d._children ? "end" : "start"; })
          .text(function(d) { return d.name; })
          .style("fill-opacity", 1);

  // Declare the links…
  var link = svg.selectAll("path.link")
          .data(links, function(d) { return d.target.id; });

  // Enter the links.
  link.enter().insert("path", "g")
          .attr("class", "link")
          .attr("d", diagonal);

}

</script>

  </body>
</html>
```

Interactive Tree Diagram

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">

    <title>Tree Example</title>

    <style>

    .node {
        cursor: pointer;
    }

    .node circle {
      fill: #fff;
      stroke: steelblue;
      stroke-width: 3px;
    }

    .node text {
      font: 12px sans-serif;
    }

    .link {
      fill: none;
      stroke: #ccc;
      stroke-width: 2px;
    }

    </style>

  </head>

  <body>

<!-- load the d3.js library -->
<script src="http://d3js.org/d3.v3.min.js"></script>

<script>

var treeData = [
  {
    "name": "Top Level",
```

```
    "parent": "null",
    "children": [
      {
        "name": "Level 2: A",
        "parent": "Top Level",
        "children": [
          {
            "name": "Son of A",
            "parent": "Level 2: A"
          },
          {
            "name": "Daughter of A",
            "parent": "Level 2: A"
          }
        ]
      },
      {
        "name": "Level 2: B",
        "parent": "Top Level"
      }
    ]
  }
];


// ************** Generate the tree diagram     *****************
var margin = {top: 20, right: 120, bottom: 20, left: 120},
    width = 960 - margin.right - margin.left,
    height = 500 - margin.top - margin.bottom;

var i = 0,
    duration = 750,
    root;

var tree = d3.layout.tree()
    .size([height, width]);

var diagonal = d3.svg.diagonal()
    .projection(function(d) { return [d.y, d.x]; });

var svg = d3.select("body").append("svg")
    .attr("width", width + margin.right + margin.left)
    .attr("height", height + margin.top + margin.bottom)
  .append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")");
```

```
root = treeData[0];
root.x0 = height / 2;
root.y0 = 0;

update(root);

d3.select(self.frameElement).style("height", "500px");

function update(source) {

  // Compute the new tree layout.
  var nodes = tree.nodes(root).reverse(),
      links = tree.links(nodes);

  // Normalize for fixed-depth.
  nodes.forEach(function(d) { d.y = d.depth * 180; });

  // Update the nodes…
  var node = svg.selectAll("g.node")
      .data(nodes, function(d) { return d.id || (d.id = ++i); });

  // Enter any new nodes at the parent's previous position.
  var nodeEnter = node.enter().append("g")
      .attr("class", "node")
      .attr("transform", function(d) {
                  return "translate(" + source.y0 + "," + source.x0 + ")"; })
      .on("click", click);

  nodeEnter.append("circle")
      .attr("r", 1e-6)
      .style("fill", function(d) {
                  return d._children ? "lightsteelblue" : "#fff"; });

  nodeEnter.append("text")
      .attr("x", function(d) {
                  return d.children || d._children ? -13 : 13; })
      .attr("dy", ".35em")
      .attr("text-anchor", function(d) {
                  return d.children || d._children ? "end" : "start"; })
      .text(function(d) { return d.name; })
      .style("fill-opacity", 1e-6);

  // Transition nodes to their new position.
  var nodeUpdate = node.transition()
      .duration(duration)
      .attr("transform", function(d) {
```

```
                return "translate(" + d.y + "," + d.x + ")"; });

nodeUpdate.select("circle")
    .attr("r", 10)
    .style("fill", function(d) {
                return d._children ? "lightsteelblue" : "#fff"; });

nodeUpdate.select("text")
    .style("fill-opacity", 1);

// Transition exiting nodes to the parent's new position.
var nodeExit = node.exit().transition()
    .duration(duration)
    .attr("transform", function(d) {
                return "translate(" + source.y + "," + source.x + ")"; })
    .remove();

nodeExit.select("circle")
    .attr("r", 1e-6);

nodeExit.select("text")
    .style("fill-opacity", 1e-6);

// Update the links…
var link = svg.selectAll("path.link")
    .data(links, function(d) { return d.target.id; });

// Enter any new links at the parent's previous position.
link.enter().insert("path", "g")
    .attr("class", "link")
    .attr("d", function(d) {
      var o = {x: source.x0, y: source.y0};
      return diagonal({source: o, target: o});
    });

// Transition links to their new position.
link.transition()
    .duration(duration)
    .attr("d", diagonal);

// Transition exiting nodes to the parent's new position.
link.exit().transition()
    .duration(duration)
    .attr("d", function(d) {
      var o = {x: source.x, y: source.y};
      return diagonal({source: o, target: o});
```

```
      })
      .remove();

  // Stash the old positions for transition.
  nodes.forEach(function(d) {
    d.x0 = d.x;
    d.y0 = d.y;
  });
}

// Toggle children on click.
function click(d) {
  if (d.children) {
    d._children = d.children;
    d.children = null;
  } else {
    d.children = d._children;
    d._children = null;
  }
  update(d);
}

</script>

  </body>
</html>
```

Force Layout Diagram

```
<!DOCTYPE html>
<meta charset="utf-8">
<script type="text/javascript" src="d3/d3.v3.js"></script>
<style>


path.link {
  fill: none;
  stroke: #666;
  stroke-width: 1.5px;
}

path.link.twofive {
  opacity: 0.25;
}

path.link.fivezero {
  opacity: 0.50;
}

path.link.sevenfive {
  opacity: 0.75;
}

path.link.onezerozero {
  opacity: 1.0;
}

circle {
  fill: #ccc;
  stroke: #fff;
  stroke-width: 1.5px;
}

text {
  fill: #000;
  font: 10px sans-serif;
  pointer-events: none;
}
```

```
</style>
<body>
<script>


// get the data
d3.csv("data/force.csv", function(error, links) {

var nodes = {};

// Compute the distinct nodes from the links.
links.forEach(function(link) {
    link.source = nodes[link.source] ||
        (nodes[link.source] = {name: link.source});
    link.target = nodes[link.target] ||
        (nodes[link.target] = {name: link.target});
    link.value = +link.value;
});

var width = 960,
    height = 500;

var force = d3.layout.force()
    .nodes(d3.values(nodes))
    .links(links)
    .size([width, height])
    .linkDistance(60)
    .charge(-300)
    .on("tick", tick)
    .start();

// Set the range
var        v = d3.scale.linear().range([0, 100]);

// Scale the range of the data
v.domain([0, d3.max(links, function(d) { return d.value; })]);

// asign a type per value to encode opacity
links.forEach(function(link) {
        if (v(link.value) <= 25) {
                link.type = "twofive";
        } else if (v(link.value) <= 50 && v(link.value) > 25) {
                link.type = "fivezero";
        } else if (v(link.value) <= 75 && v(link.value) > 50) {
                link.type = "sevenfive";
        } else if (v(link.value) <= 100 && v(link.value) > 75) {
                link.type = "onezerozero";
```

```
        }
});

var svg = d3.select("body").append("svg")
    .attr("width", width)
    .attr("height", height);

// build the arrow.
svg.append("svg:defs").selectAll("marker")
    .data(["end"])
  .enter().append("svg:marker")
    .attr("id", String)
    .attr("viewBox", "0 -5 10 10")
    .attr("refX", 15)
    .attr("refY", -1.5)
    .attr("markerWidth", 6)
    .attr("markerHeight", 6)
    .attr("orient", "auto")
  .append("svg:path")
    .attr("d", "M0,-5L10,0L0,5");

// add the links and the arrows
var path = svg.append("svg:g").selectAll("path")
    .data(force.links())
  .enter().append("svg:path")
    .attr("class", function(d) { return "link " + d.type; })
    .attr("marker-end", "url(#end)");

// define the nodes
var node = svg.selectAll(".node")
    .data(force.nodes())
  .enter().append("g")
    .attr("class", "node")
    .on("click", click)
    .on("dblclick", dblclick)
    .call(force.drag);

// add the nodes
node.append("circle")
    .attr("r", 5);

// add the text
node.append("text")
    .attr("x", 12)
    .attr("dy", ".35em")
    .text(function(d) { return d.name; });
```

```
// add the curvy lines
function tick() {
    path.attr("d", function(d) {
        var dx = d.target.x - d.source.x,
            dy = d.target.y - d.source.y,
            dr = Math.sqrt(dx * dx + dy * dy);
        return "M" +
            d.source.x + "," +
            d.source.y + "A" +
            dr + "," + dr + " 0 0,1 " +
            d.target.x + "," +
            d.target.y;
    });

    node
        .attr("transform", function(d) {
                    return "translate(" + d.x + "," + d.y + ")"; });
}

// action to take on mouse click
function click() {
    d3.select(this).select("text").transition()
        .duration(750)
        .attr("x", 22)
        .style("fill", "steelblue")
        .style("stroke", "lightsteelblue")
        .style("stroke-width", ".5px")
        .style("font", "20px sans-serif");
    d3.select(this).select("circle").transition()
        .duration(750)
        .attr("r", 16)
        .style("fill", "lightsteelblue");
}

// action to take on mouse double click
function dblclick() {
    d3.select(this).select("circle").transition()
        .duration(750)
        .attr("r", 6)
        .style("fill", "#ccc");
    d3.select(this).select("text").transition()
        .duration(750)
        .attr("x", 12)
        .style("stroke", "none")
        .style("fill", "black")
```

```
        .style("stroke", "none")
        .style("font", "10px sans-serif");
}

});


</script>
</body>
</html>
```

Bullet Chart

```
<!DOCTYPE html>
<meta charset="utf-8">
<style>


body {
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
  margin: auto;
  padding-top: 40px;
  position: relative;
  width: 800px;
}

button {
  position: absolute;
  right: 40px;
  top: 10px;
}

.bullet { font: 10px sans-serif; }
.bullet .marker { stroke: #000; stroke-width: 2px; }
.bullet .tick line { stroke: #666; stroke-width: .5px; }
.bullet .range.s0 { fill: #eee; }
.bullet .range.s1 { fill: #ddd; }
.bullet .range.s2 { fill: #ccc; }
.bullet .measure.s0 { fill: steelblue; }
.bullet .title { font-size: 14px; font-weight: bold; }
.bullet .subtitle { fill: #999; }


</style>
<button>Update</button>
<script type="text/javascript" src="d3/d3.v3.js"></script>
<script src="js/bullet.js"></script>
<script>
```

```
var margin = {top: 5, right: 40, bottom: 20, left: 120},
    width = 800 - margin.left - margin.right,
    height = 50 - margin.top - margin.bottom;

var chart = d3.bullet()
    .width(width)
    .height(height);

d3.json("data/cpu1.json", function(error, data) {
  var svg = d3.select("body").selectAll("svg")
      .data(data)
    .enter().append("svg")
      .attr("class", "bullet")
      .attr("width", width + margin.left + margin.right)
      .attr("height", height + margin.top + margin.bottom)
    .append("g")
      .attr("transform", "translate(" + margin.left + "," + margin.top + ")")
      .call(chart);

  var title = svg.append("g")
      .style("text-anchor", "end")
      .attr("transform", "translate(-6," + height / 2 + ")");

  title.append("text")
      .attr("class", "title")
      .text(function(d) { return d.title; });

  title.append("text")
      .attr("class", "subtitle")
      .attr("dy", "1em")
      .text(function(d) { return d.subtitle; });

  d3.selectAll("button").on("click", function() {
    svg.datum(randomize).call(chart.duration(1000));
  });
});

function randomize(d) {
  if (!d.randomizer) d.randomizer = randomizer(d);
  d.markers = d.markers.map(d.randomizer);
  d.measures = d.measures.map(d.randomizer);
  return d;
}

function randomizer(d) {
  var k = d3.max(d.ranges) * .2;
```

```
  return function(d) {
    return Math.max(0, d + k * (Math.random() - .5));
  };
}


</script>
</body>
```

Map with zoom / pan and cities

```
<!DOCTYPE html>
<meta charset="utf-8">
<style>


path {
  stroke: white;
  stroke-width: 0.25px;
  fill: grey;
}


</style>
<body>
<script type="text/javascript" src="d3/d3.v3.js"></script>
<script src="js/topojson.v0.min.js"></script>
<script>


var width = 960,
    height = 500;

var projection = d3.geo.mercator()
    .center([0, 5 ])
    .scale(900)
    .rotate([-180,0]);

var svg = d3.select("body").append("svg")
    .attr("width", width)
    .attr("height", height);

var path = d3.geo.path()
    .projection(projection);

var g = svg.append("g");

// load and display the World
d3.json("json/world-110m2.json", function(error, topology) {
    g.selectAll("path")
      .data(topojson.object(topology, topology.objects.countries)
          .geometries)
    .enter()
      .append("path")
      .attr("d", path)
```

```
    // load and display the cities
    d3.csv("data/cities.csv", function(error, data) {
        g.selectAll("circle")
           .data(data)
           .enter()
           .append("circle")
           .attr("cx", function(d) {
                   return projection([d.lon, d.lat])[0];
           })
           .attr("cy", function(d) {
                   return projection([d.lon, d.lat])[1];
           })
           .attr("r", 5)
           .style("fill", "red");
    });

});

// zoom and pan
var zoom = d3.behavior.zoom()
    .on("zoom",function() {
        g.attr("transform","translate("+
            d3.event.translate.join(",")+")scale("+d3.event.scale+")");
        g.selectAll("path")
            .attr("d", path.projection(projection));
        g.selectAll("circle")
            .attr("d", path.projection(projection));
  });

svg.call(zoom)


</script>
</body>
</html>
```