D3

Tips and Tricks

Because Life's too Short to Learn All the Things

d3noob.org

D3 Tips and Tricks

Because Life's too Short to Learn All the Things

Malcolm Maclean

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

This version was published on 2013-03-20

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.

Contents

Acknowledgements

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.

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.

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.

Mike Dewar (*Getting Started with D3*), Scott Murray (*Interactive Data Visualization for the Web*) and Sebastian Gutierrez (dashingd3js.com[2]) lead the pack for providing high quality reference material for learning D3. Many thanks gentlemen.

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

Lastly, 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] http://www.dashingd3js.com/

What is d3.js?

d3.js[3] (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 it's 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[4] who has not just spent time writing the code, but writing the documentation[5] for D3 as well. There is an extensive community of supporters who also contribute to the code, provide technical support[6] online[7] and generally have fun creating amazing visualizations[8]. Their contribution is extrodinary (you only have to look at the work of Jason Davies to be amazed).

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

Introduction

OK, weird sensation…

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 and I needed to do a lot of searching and trialling by 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 to 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 others good 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 hyper-links 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 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 some cases the return on the investment of the training that would require me to do things in a particular (perhaps best practices) way wasn't justified. And 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 flowerier that might be to everyone's liking, but there you go, try to be brave :-)

What do you need to get started?

Let's be frank, my Grandmother will never put together 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.

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

DON'T FREAK OUT!

First things first. This isn't rocket science. It's just the 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[9] for a great overview. Just remember... 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[10] 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 it's fanciness. In fact, D3 is a JavaScript Library, so that's like 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[11]

[9] http://en.wikipedia.org/wiki/HTML

[10] http://en.wikipedia.org/wiki/JavaScript

[11] 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[12] (everyone appears to call them 'Style Sheets' or 'CSS') is what's known as 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 noob's like myself will find it easier, but where possible try and use examples that include what look like logical CSS structures)

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

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…

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[13] 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[14] and present the resulting graphics on bl.ocks.org[15]. 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.

[13] https://github.com/mbostock/d3/wiki
[14] https://github.com/about
[15] http://bl.ocks.org/

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.

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 as both a relatively simple language to learn, but also 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;
});
```

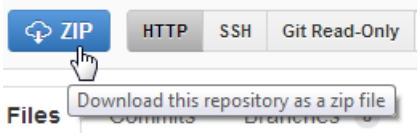
Infinity easier to use. Trust me.

There are plenty of editors that will do the trick. I have a preference for Geany[16], mainly because it's what I started with and it grew on me :-).

Getting D3

Luckily this is pretty easy.

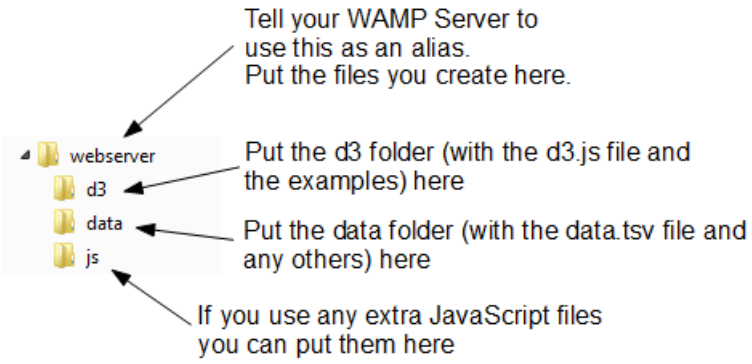
Go to the D3 repository on github[17] and download the entire repository by clicking on the 'ZIP' button.



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

[16] http://www.geany.org/
[17] https://github.com/mbostock/d3

- 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[18] 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[19] 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[20] 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[21].

[18] https://github.com/mbostock/d3/wiki/Gallery
[19] https://github.com/mbostock/d3/wiki
[20] https://github.com/mbostock/d3
[21] https://groups.google.com/forum/?fromgroups#!forum/d3-js

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 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[22] is predominantly a code repository and version control site. It is highly regarded for its technical acumen and provide 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 it's 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.

[22] https://github.com/

bl.ocks.org

bl.ocks.org[23] 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[24] 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 hosing solution, it is excellent and well worth the time to get to grips with.

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 a 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 meany 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*[25]" 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*[26]" 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)

[23] http://bl.ocks.org/

[24] https://groups.google.com/forum/?fromgroups=#!topic/d3-js/g7BxBMUZP8o

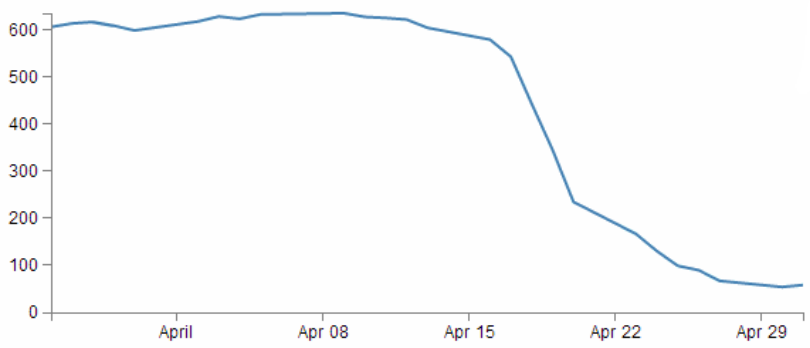
[25] http://shop.oreilly.com/product/0636920025429.do

[26] 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 appendicies as 'Simple Graph'.).

Here's what the basic graph looks like;



Basic Graph

And here's the code that makes it happen;

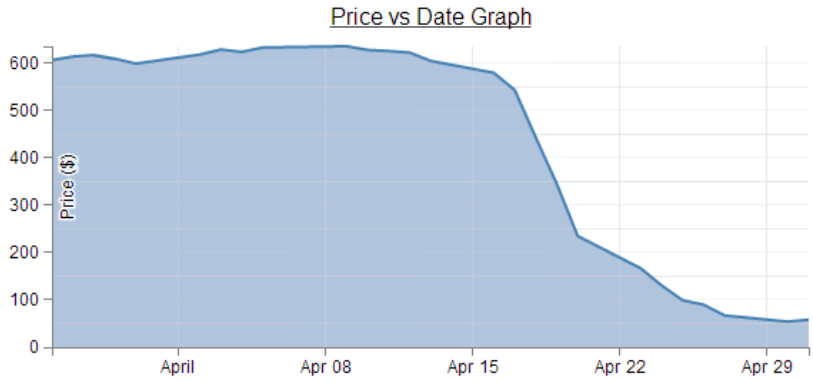
```
<!DOCTYPE html>
<meta charset="utf-8">
<style>


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>
<script type="text/javascript" src="d3/d3.v3.js"></script>

<script>


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.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; })]);

    svg.append("path")              // Add the valueline path.
```

```
        .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);

});


</script>
</body>
```

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;



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 type="text/javascript" src="d3/d3.v3.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 type="text/javascript" src="d3/d3.v3.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 stored in a folder called d3 which itself is in the same directory as the main html file. The D3 file is actually called d3.v3.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 has recently been superseded.

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;
}
```

So Cascading Style Sheets give you control over the look / feel / presentation of the 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 selector 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...

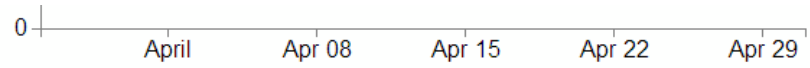
x Axis with 12px Arial

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...



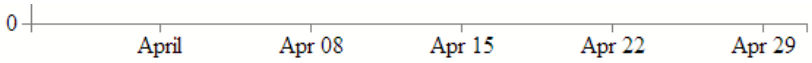
x Axis with 16px Arial

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;

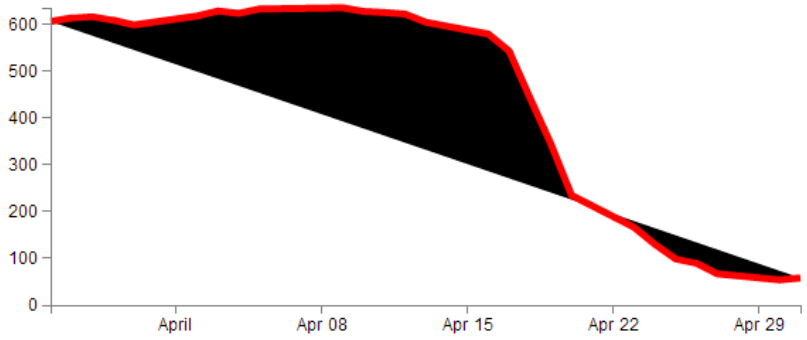
```
path {
    stroke: steelblue;
    stroke-width: 2;
    fill: none;
}
```

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;
}
```



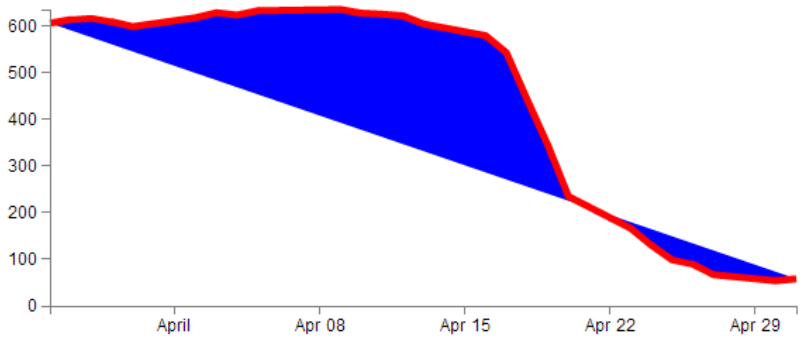
Filling of a path

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.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; })]);

    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[27], 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 array 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.

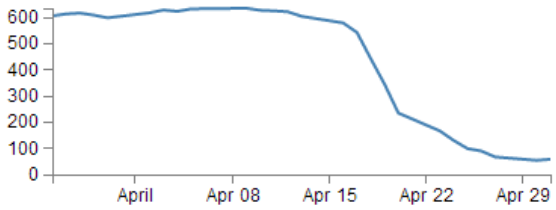
Obviously the same treatment is given to height.

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.

[27] 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.tsv("data/data.tsv", 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.

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[28]).
- **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.

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

- **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[29] page. In this particular script we will look at the tsv request method.

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.tsv("data/data.tsv", 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.tsv request (`d3.tsv`) and then the function is pointed to the data file that should be loaded (`data/data.tsv`). This is referred to as the 'url' (unique resource locator) of the file. In this case the file is stored locally, but the url could just as easily point to a file somewhere on the Internet.

The format of the data in the data.tsv 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 tab as are each subsequent dates and numbers. Hence the 'tab 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').

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

```
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 tsv 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 'data' 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 tsv 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 (tab separated values) and ensured that it is formatted in a way that the rest of the script can use it correctly.

Now, the astute amongst you will have noticed that in the first line of that block of code (`d3.tsv("data/data.tsv", 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.tsv 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[30].

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 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 to D3 to play to it's strengths. And one of those is being able to adjust dynamically with variable time values.

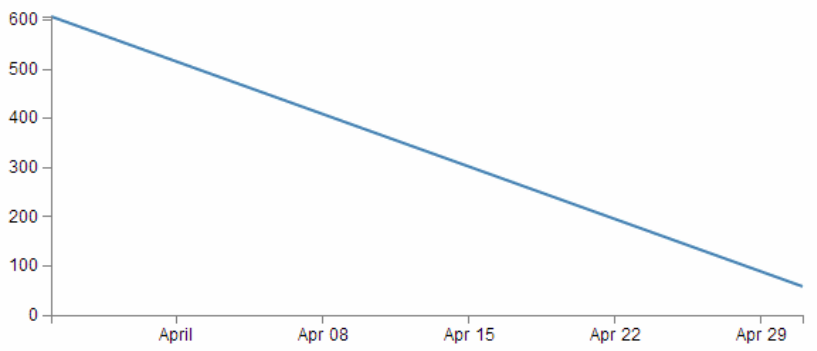
Time for a little demonstration.

We will change our data.tsv 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;

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



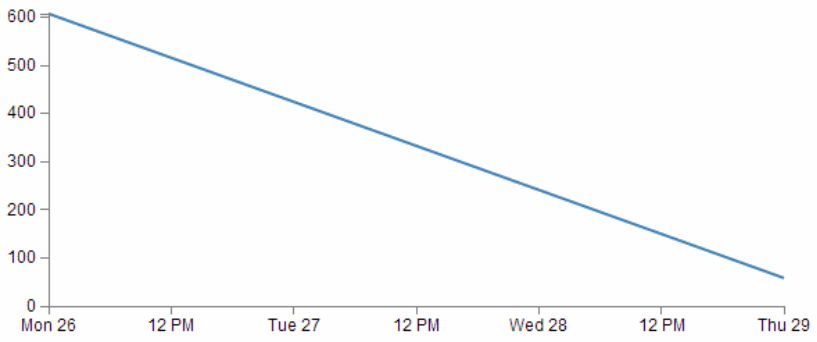
Simple line graph

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.tsv 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.



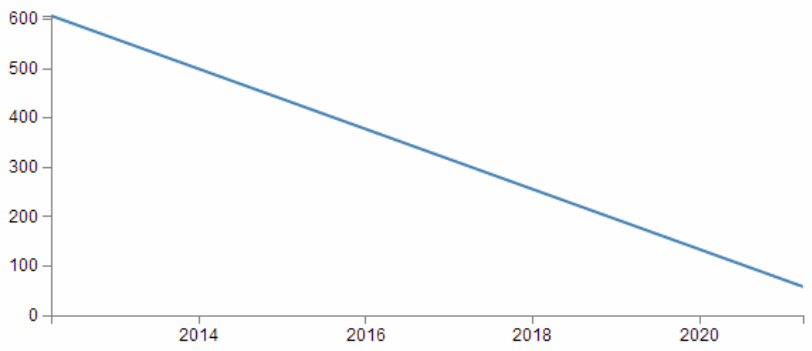
Simple line graph over three days

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 filled 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 with century as a decimal number.

If we look at a subset of the data from the data.tsv 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[a]'. 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.

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

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 tsv 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.

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.

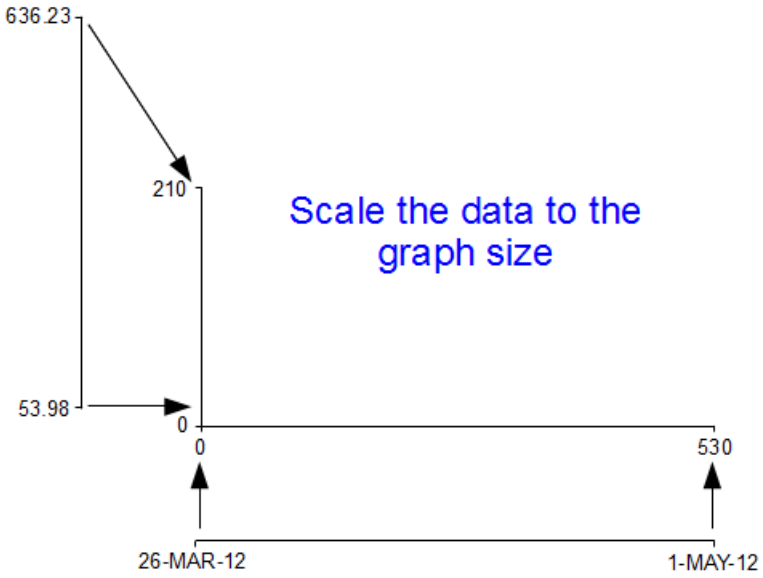
```
var x = d3.time.scale().range([0, width]);
```

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;



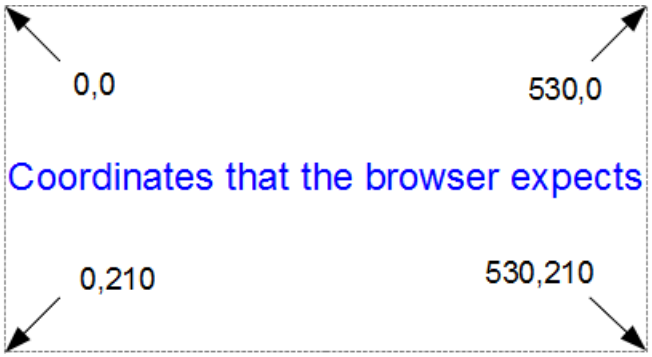
Scaling the data to the graph size

I know, I know, it's a little misleading because nowhere have we atually 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;

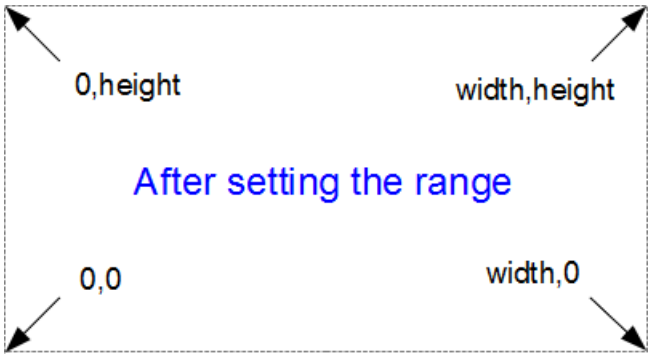


Coordinates that the browser expects

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.tsv 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.

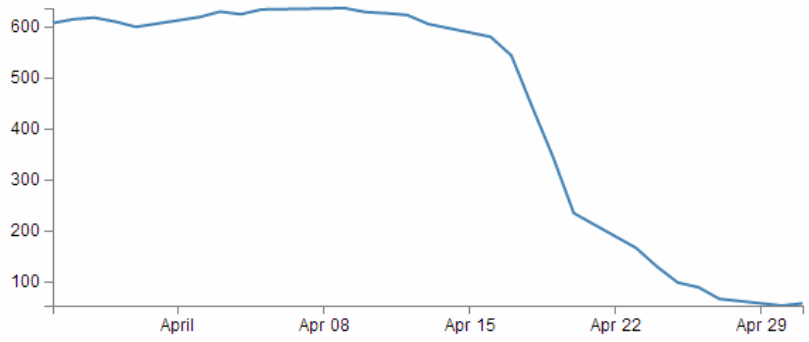
Let's try a small experiment. Let's change the y axis domain to use the .extent function (the same way the x axis does) to see what it produces.

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…



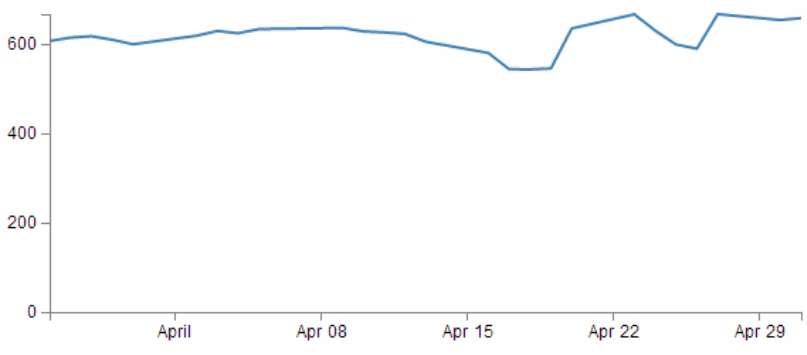
Graph using .extent for data values

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.

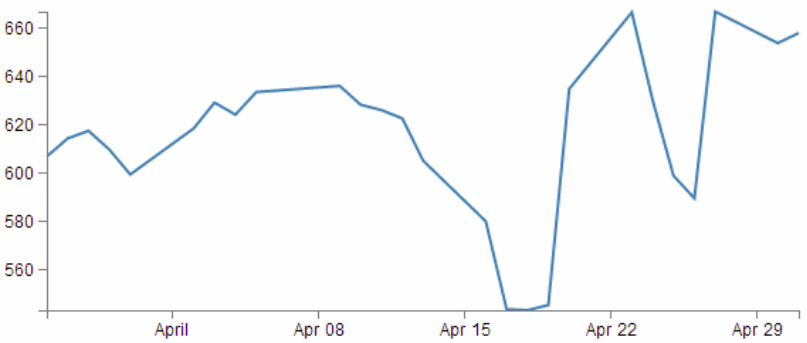
For instance, if I change the data.tsv file so that the values are represented like the following;



Concentrated data range graph

Then it kind of looses 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);
```

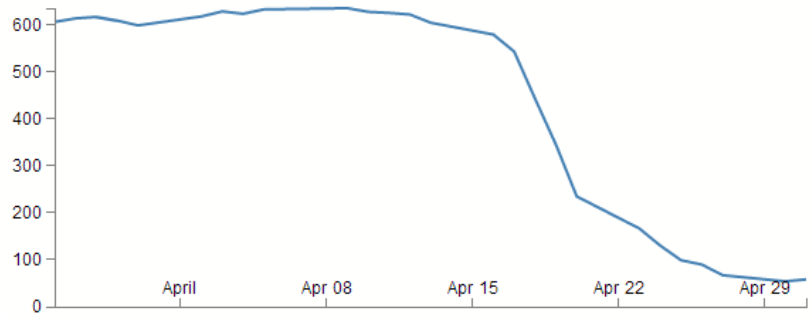
I've included both the x and y axes because they carry out the formatting in very similar ways. It's worth noting that this is not the point where the axes get drawn. That occurs later in the piece where the data.tsv file has been loaded as 'data'.

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 setup 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;

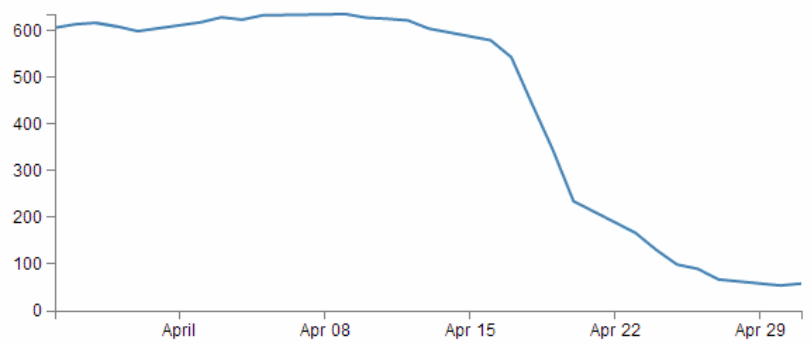


x axis orientated to top

Well, I hope you didn't see that coming, because I didn't. It transpires that what we're talking about there 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.

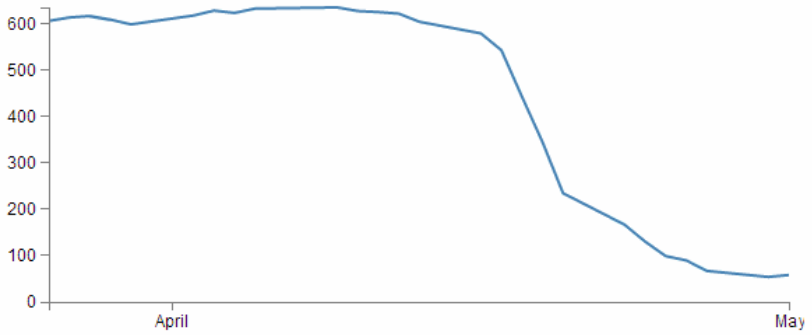
The next part (`.ticks(5)`) sets the number of ticks on the axis. Hopefully you just did a quick count across the bottom of the previous graph and went "Yep, five ticks. Spot on". Well done if you did, but there's a little bit of a sneaky trick up D3's sleeve with the number of ticks on a graph axis.

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[31] of time intervals that D3 will consider when setting automatic ticks on a time based axis;

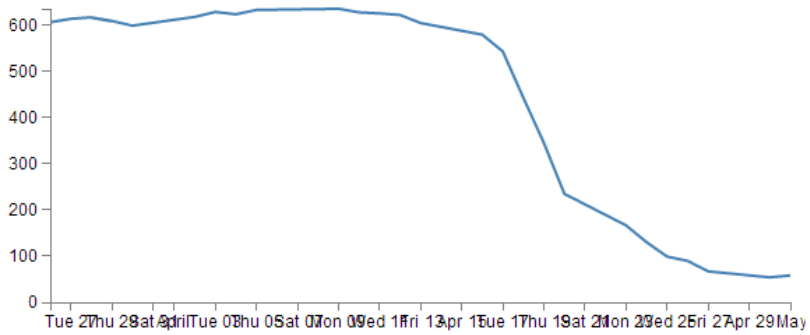
- 1-, 5-, 15and 30-second.
- 1-, 5-, 15and 30-minute.
- 1-, 3-, 6and 12-hour.
- 1 and 2-day.
- 1-week.
- 1 and 3-month.

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

- 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.



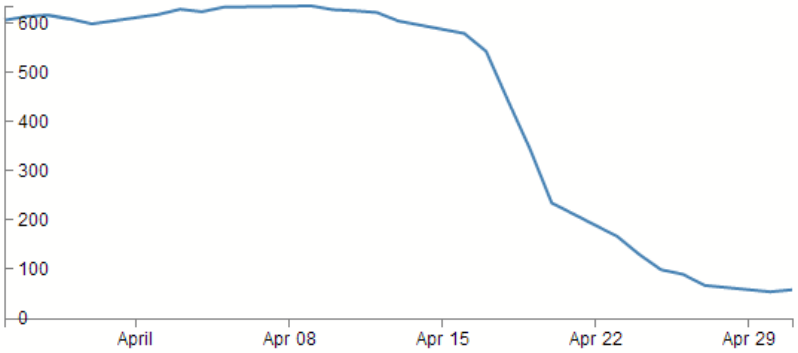
Ten ticks on the x axis

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.

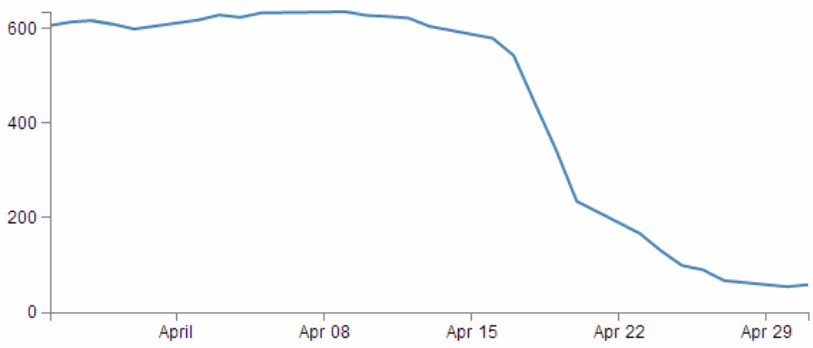


y axis right orientated

Nope. Not a pretty sight.

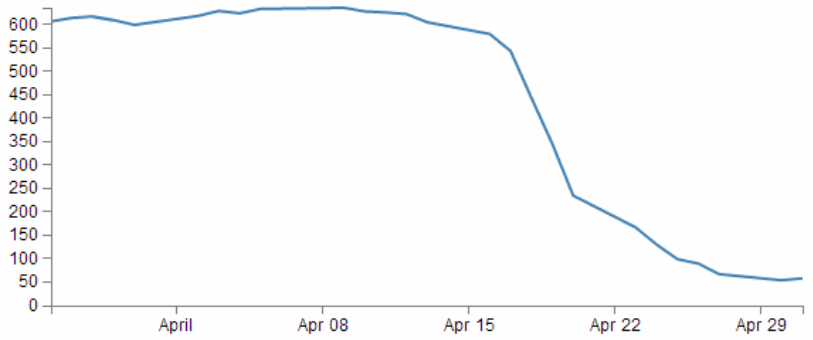
What about the number of ticks? Well this scale is quite different to the x axis. Formatting the dates using logical separators (weeks, months) was tricky, but with standard numbers, it should be a little easier. In fact, there's a fair chance that you've already had a look at the y axis and seen that 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[32]. This function uses assessor functions to store the appropriate information in the right area and in the case above they use the x and y assessors (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 our 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 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.

```
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?

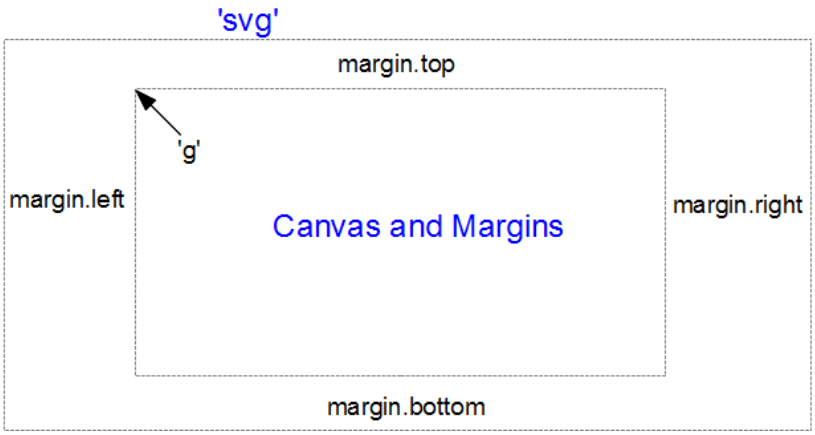
Well D3 need to be able to have a space defined for it to draw things. And when you define the space it's going to use, you can also give the space you're going to use a name, attributes and positions within that space a designation.

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.

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

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[33]). 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.

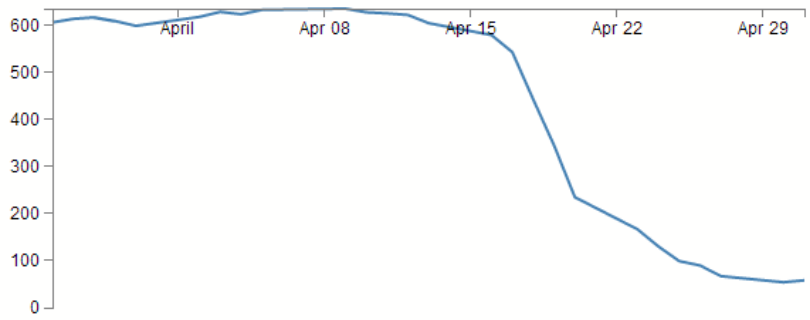
So both axes start by being appended to the "g" group. Then each has its own classes applied for styling via CSS. If you recall from earlier, they look a little like this;

[33] 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 as 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.

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 :-).

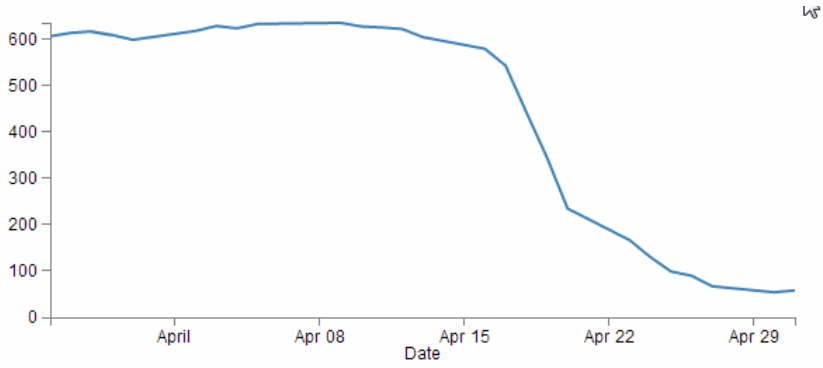
This is the code we're going to add to the simple line graph script;

```
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)[34]. 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.

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.

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

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.

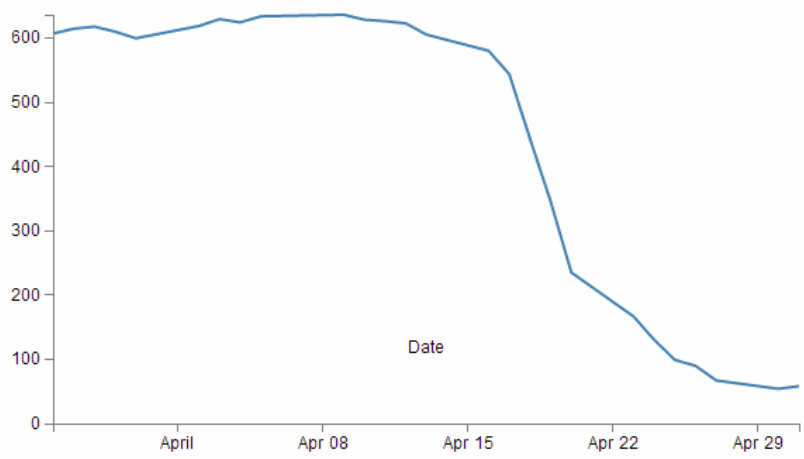
Here's an example. If I decide that I would prefer to increase the height of the graph by editing the line here;

```
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;

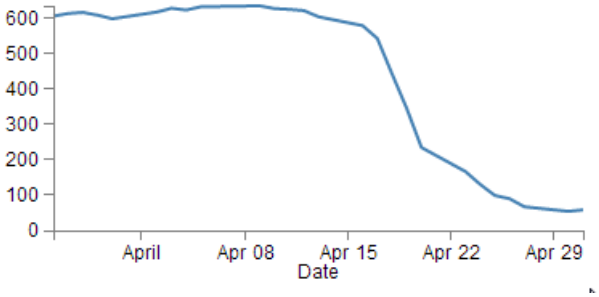
```
.attr("x", width / 2 )
.attr("y",  height + margin.bottom)
```

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 its 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).

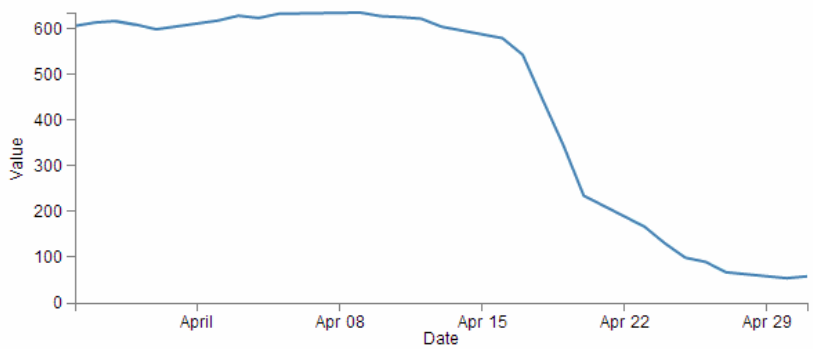
So, that's the x axis label. Time to do the y axis. The code we're going to use looks like this;

```
    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 test element to our canvas (`svg.append("text")`).

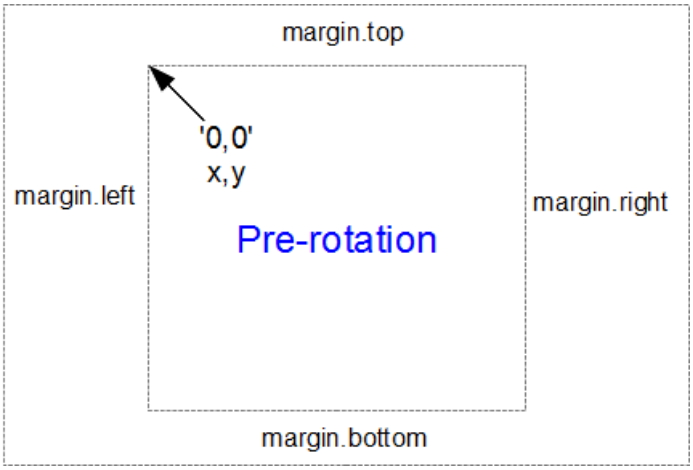
Then things get interesting.

```
        .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))
```

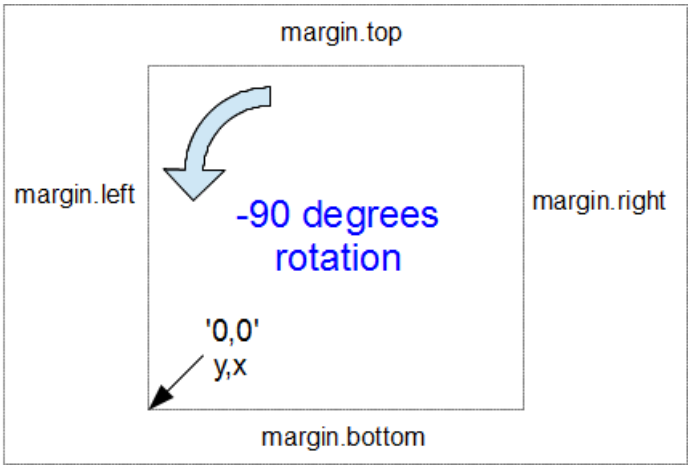
Let's get graphical to illustrate how this works;



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 is 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[35]. 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 and they make sure 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.

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

How to add a title to your graph

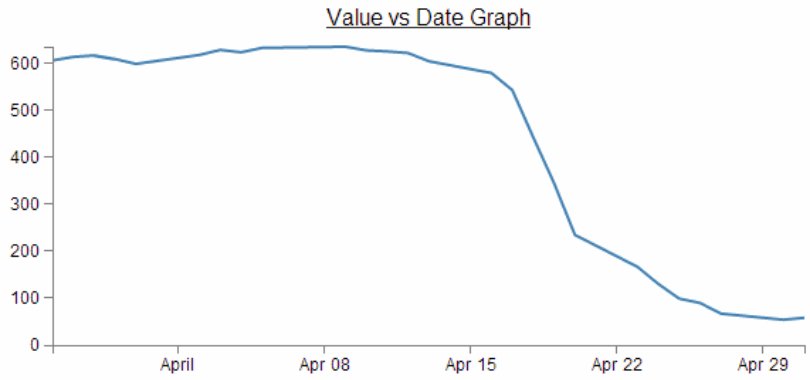
If you've read through the adding the axis labels section most of this will come as no surprise.

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.

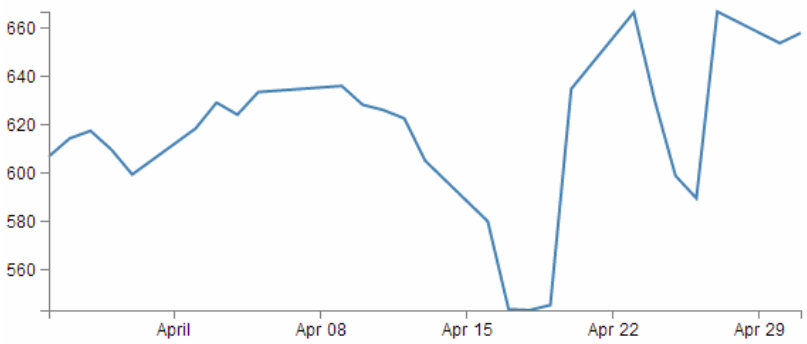
Smoothing out graph lines

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 coordinated 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.

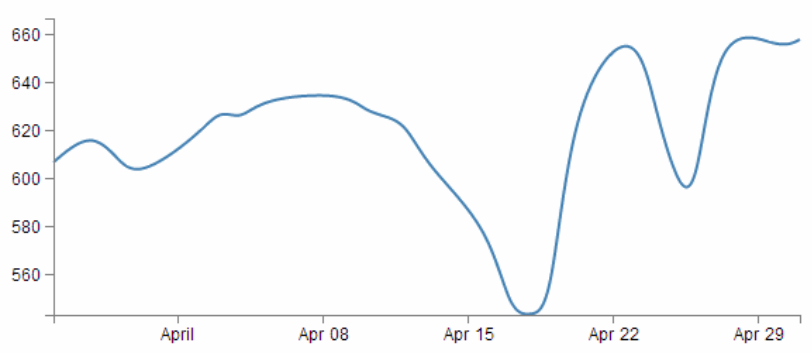


Expanded values for a narrow range

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.

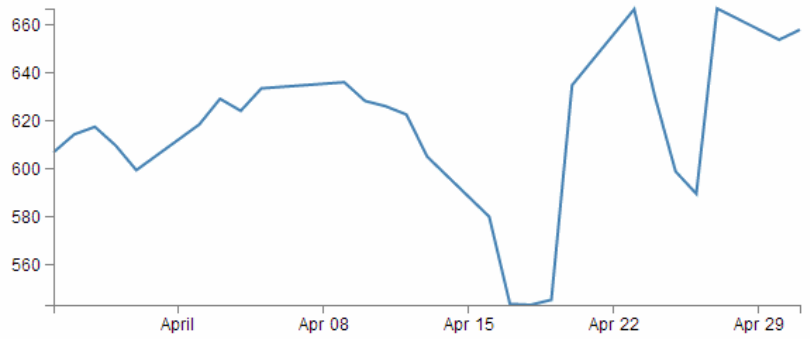
Here's the list of available options and for more about them head on over to the D3 wiki[36] and look for 'line.interpolate'.

- 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.

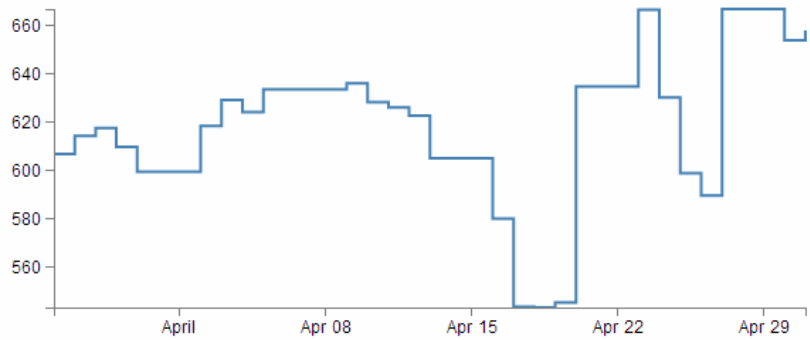
[36] https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-line_interpolate

- 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.

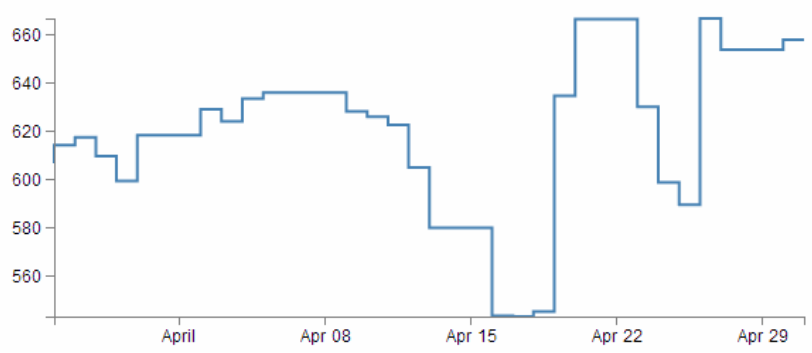
Because in the course of writing this I took an opportunity to play with each of them, I was pleasantly surprised to see some of the effects and it seems like a shame to deprive the reader of the same joy :-). So at the risk of deforesting the planet (so I hope you are reading this in electronic format)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



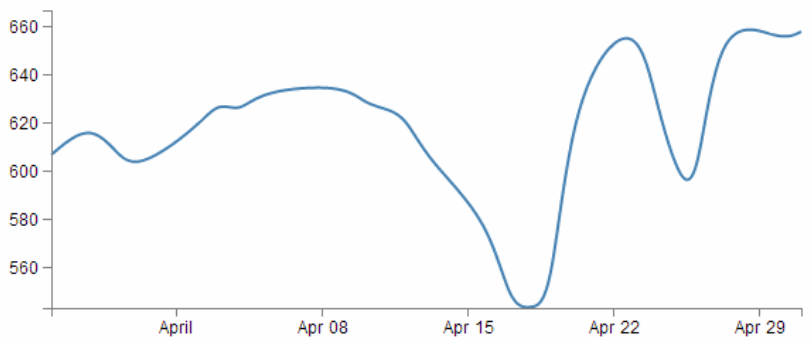
Smoothing using "linear"



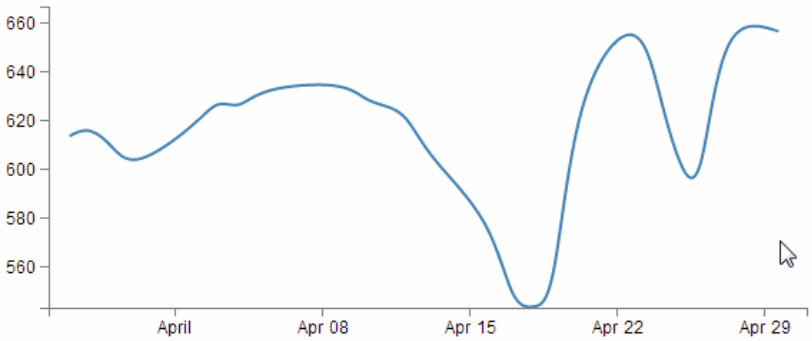
Smoothing using "step-before"



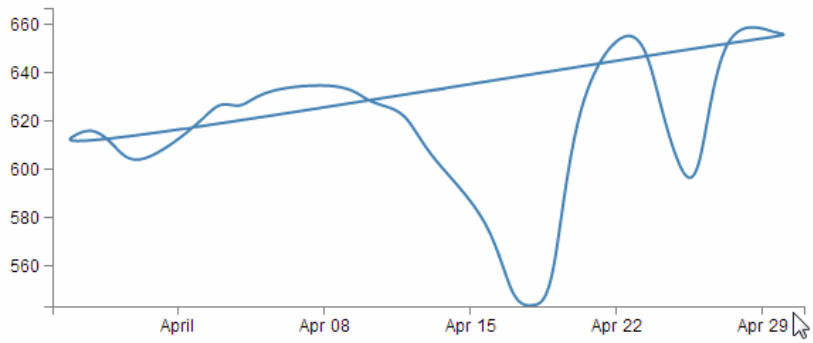
Smoothing using ”step-after”



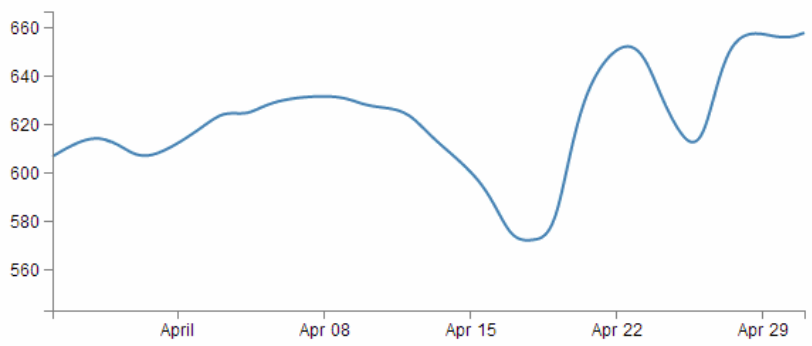
Smoothing using ”basis”



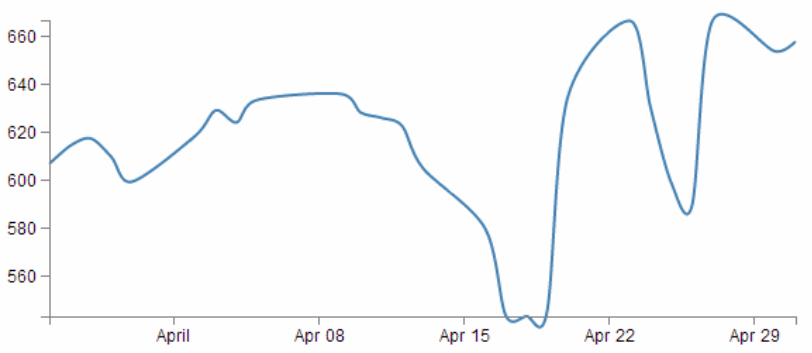
Smoothing using ”basis-open”



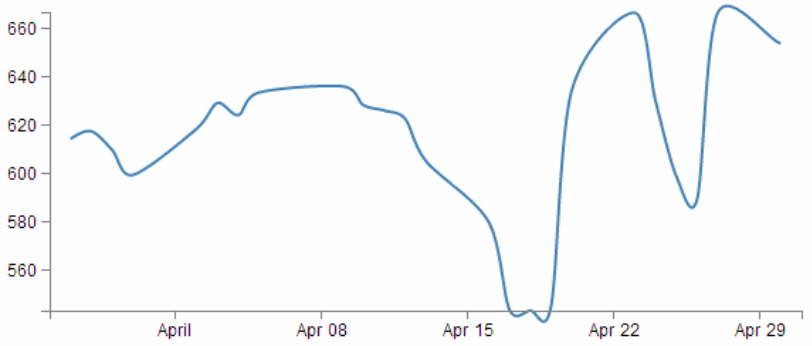
Smoothing using ”basis-closed”



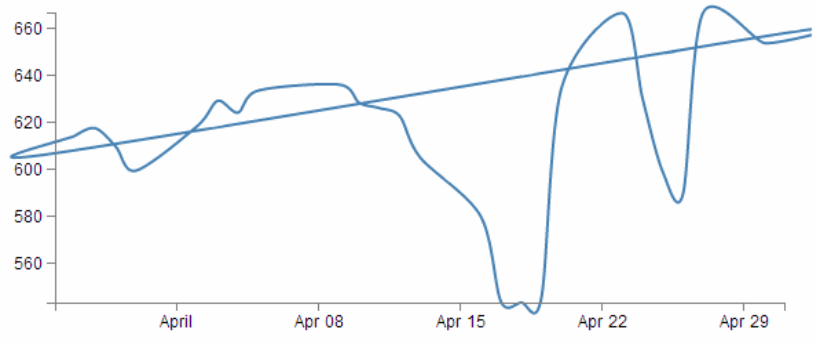
Smoothing using ”bundle”



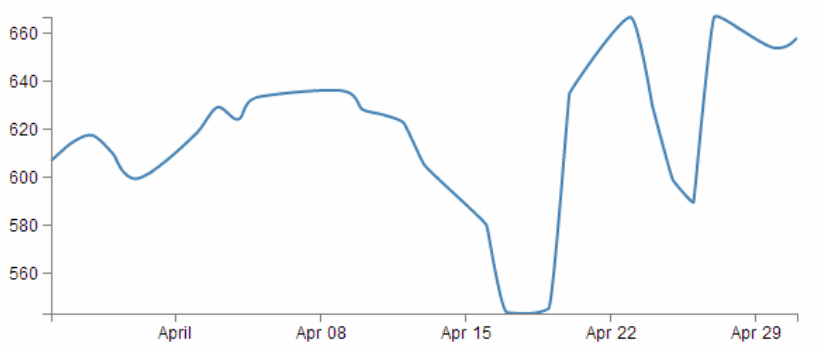
Smoothing using ”cardinal”



Smoothing using ”cardinal-open”



Smoothing using ”cardinal-closed”



Smoothing using ”monotone”

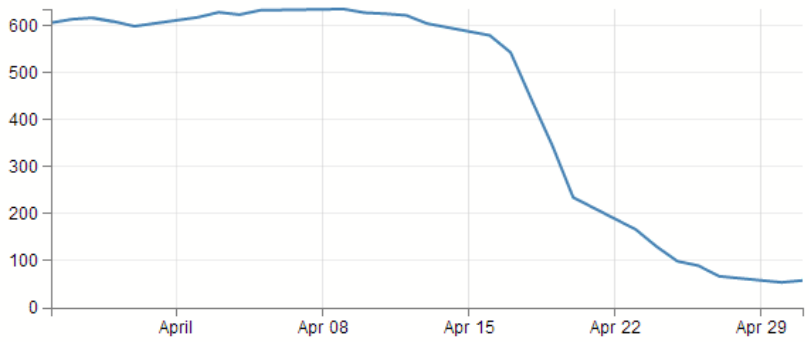
So, over to you to decide which format of interpolation is going to suit your data best:-).

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[a]. 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 :-)

[a] http://dealloc.me/2011/06/24/d3-is-not-a-graphing-library.html

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[37] 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.

[37] http://knowyourmeme.com/memes/longcat

The grid line CSS

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

```
.grid .tick {
    stroke: lightgrey;
    opacity: 0.7;
}
.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) and the opacity (transparency) of the lines.

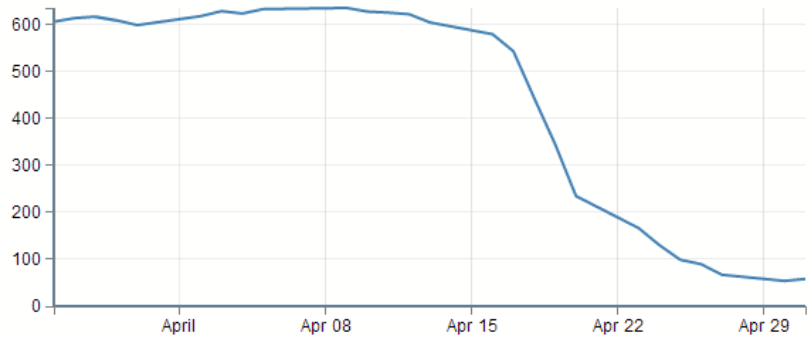
```
    stroke: lightgrey;
    opacity: 0.7;
```

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).

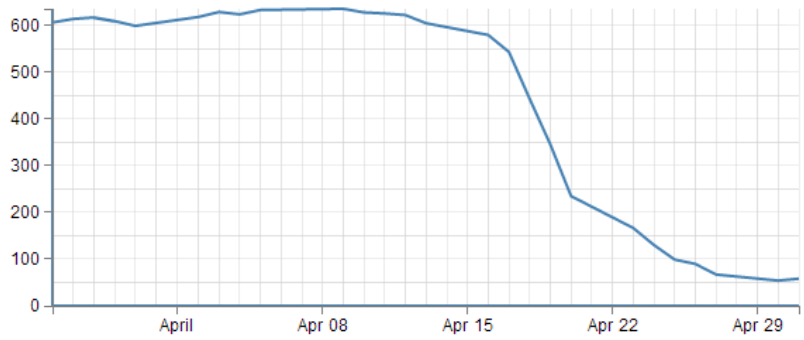
A good spot to place the code is just before we load the data with the d3.tsv

```
        //   <== Put the functions here!
// Get the data
d3.tsv("data/data.tsv", function(error, data) {
    data.forEach(function(d) {
        d.date = parseDate(d.date);
        d.close = +d.close;
    });ticks(5)
}
```

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 each will be fed back to the drawing process when called. Both make sure that is uses the d3.svg.axis function and then they set individual attributes which make sense.

The 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, if we think back to when we were setting up the axis for the basic graph and we mess about, seeing how many 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[38] is that for the .tickSize function, the following is the format.

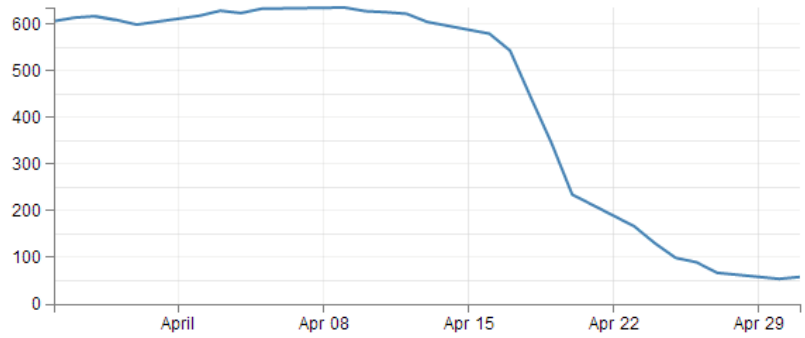
[38] https://github.com/mbostock/d3/wiki/SVG-Axes#wiki-tickSize

```
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 :-)

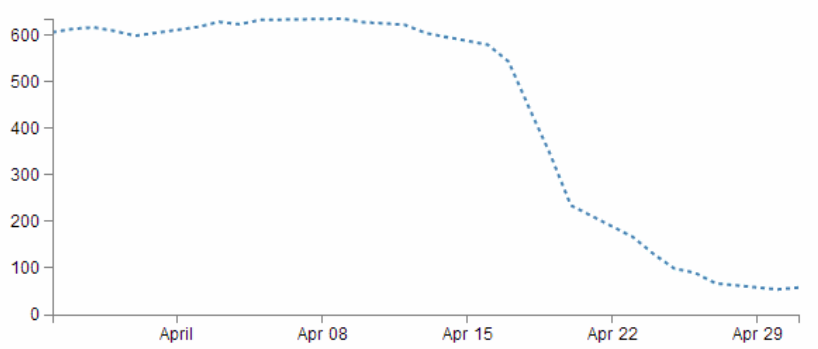
One of the best parts about it is that they're so simple to do!

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.

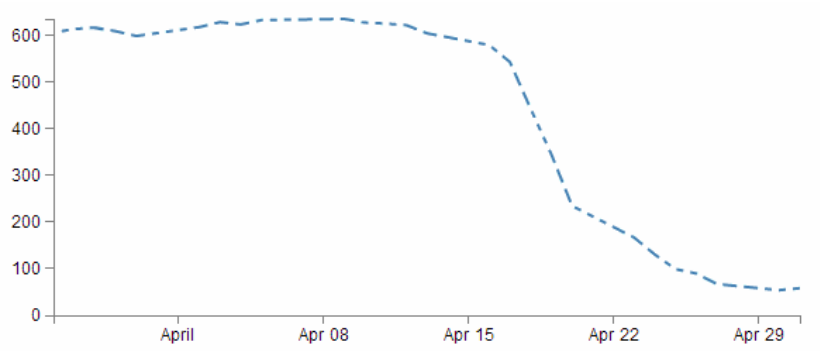
Essentially they describe the on length and off length of the line. So `"3, 3"` translates to 3 pixels (or whatever they are) on and 3 pixels off. Then it repeats. Simple eh?

So, experiment time :-)

What would the following represent?

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

Try not to cheat...

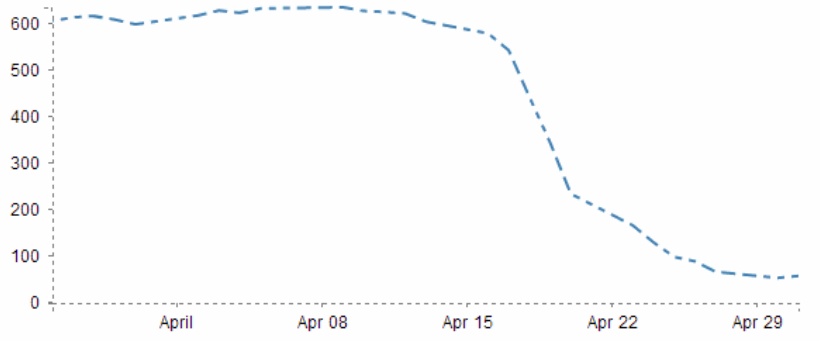


Dashed lines for fun

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. Some times you've just got to go with a fill.

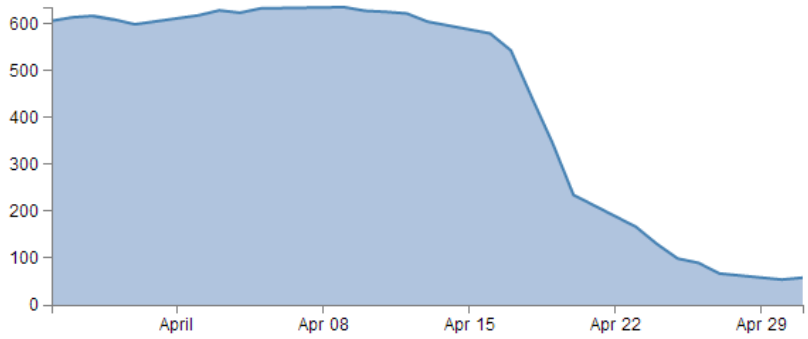
Filling an area with a solid colour isn't too hard. I mean we did it by mistake back a few pages when we were trying to draw a line.

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 it is done in three sections;

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

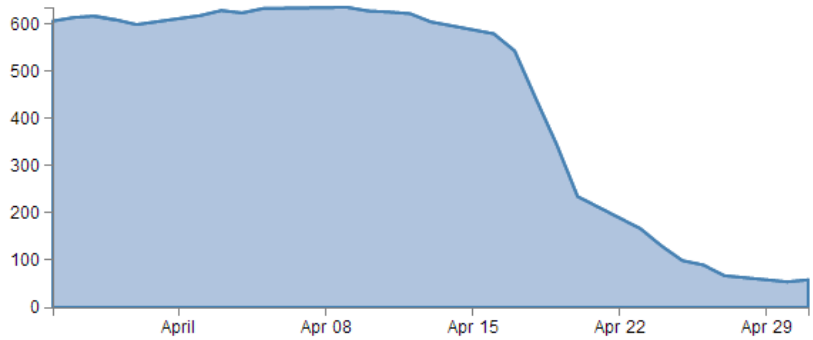
This is pretty straight forward and only consists of two rules;

```
.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[39]

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); });
```

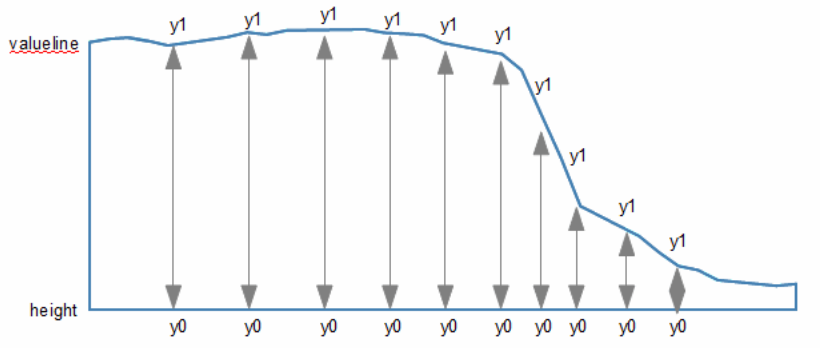
You will notice it looks *INCREDIBLY* similar to the valueline function definition. That's because; while the line definition describes drawing a line that connects a set of coordinates, I imagine the area definition describes drawing two lines that share the same x coordinates, but simultaneously draws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

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

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;



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;

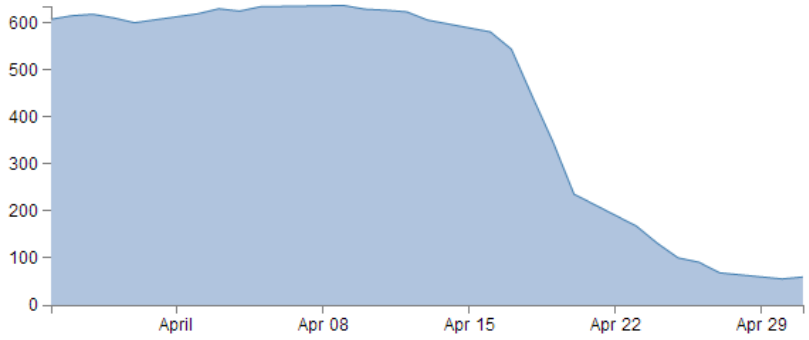
```
svg.append("path")
    .datum(data)
    .attr("class", "area")
    .attr("d", area);
```

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?

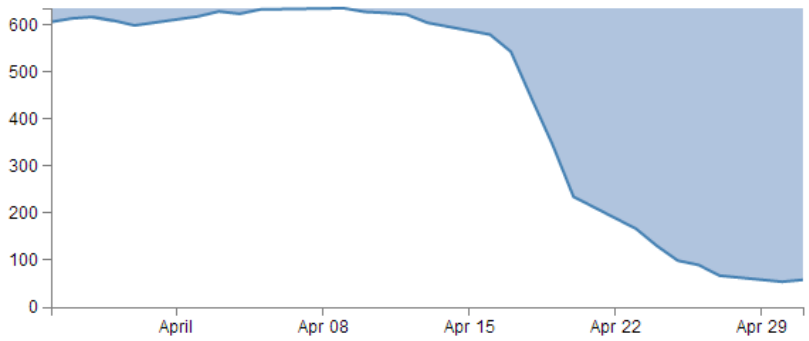
Well, remember the code that defined the area?

```
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.

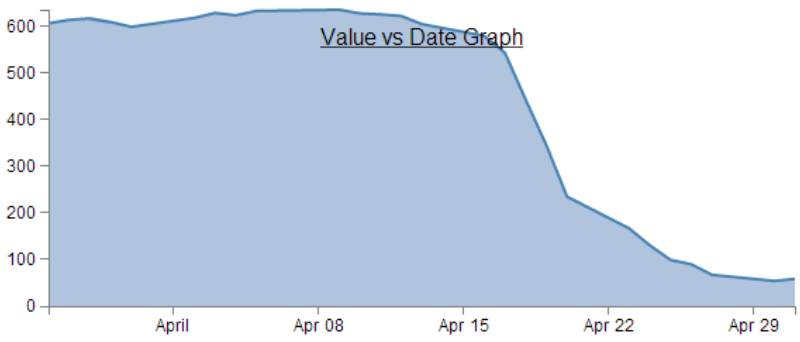
I've deliberately positioned this particular tip to follow the 'filling an area' description because it provides an opportunity to demonstrate the principle to slightly better effect.

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;



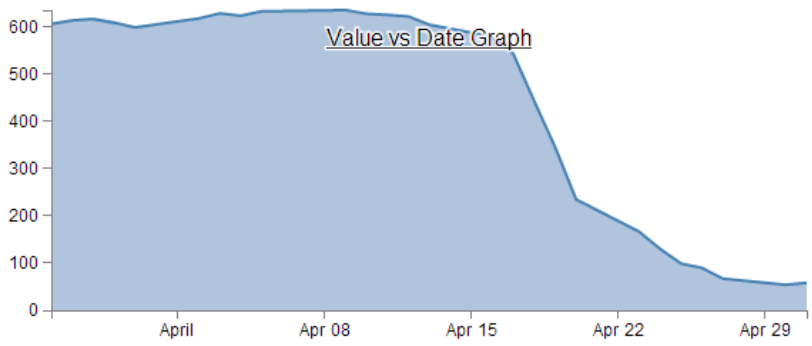
Title lost in the area fill

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!

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.tsv file. In fact it looks a little like this (apologies in advance for the big ugly block of data);

```
date        close   open
1-May-12    58.13   34.12
30-Apr-12   53.98   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 tab (hence .tsv (Tab 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 let's call it data2.tsv.

We will be 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.tsv' to reflect the new name.

```
d3.tsv("data/data2.tsv", 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); });
```

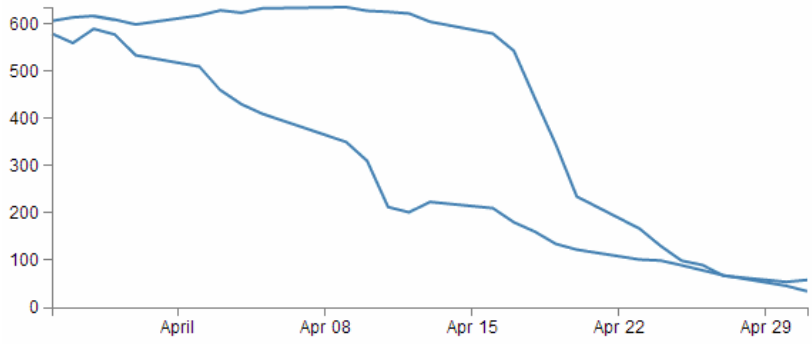
You should notice that this block is identical to the block that sets up the function for the first line, except this one is called (imaginatively) valueline2. We should put it directly after the block that sets up the function for valueline.

The second block draws our new line;

```
    svg.append("path")          // Add the valueline2 path.
        .attr("class", "line")
        .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;



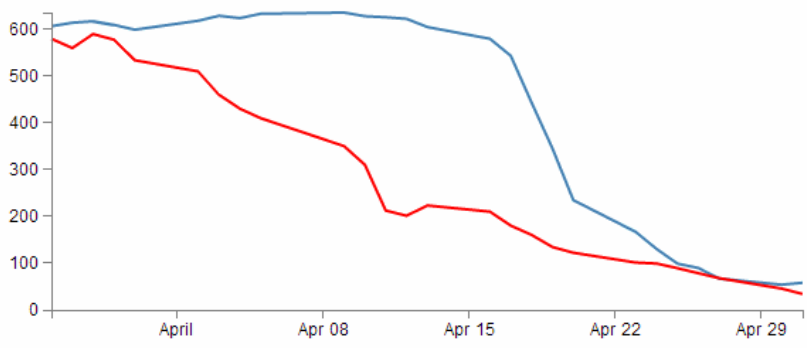
Two lines, but the same colour

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.
        .attr("class", "line")
        .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.

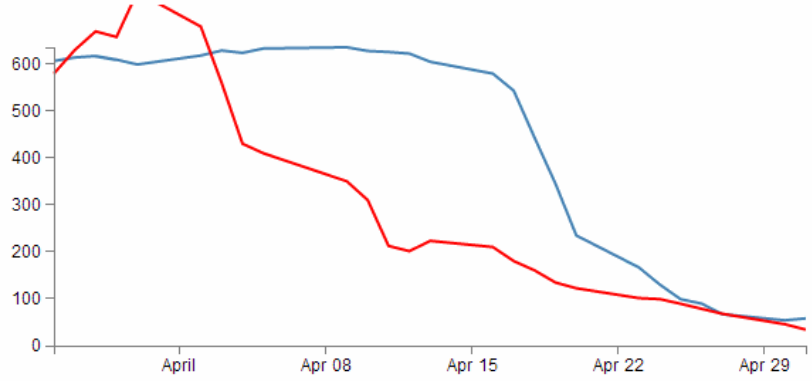
The first mistake we made was not ensuring that our variable `"d.open"` is being treated as a number or a string. We're fortunate in this case that it is, but this can't always be assumed. So, this is an easy fix and we just need to put the following (indicated line) in our code;

```
// Get the data
d3.tsv("data/data.tsv", 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[40];

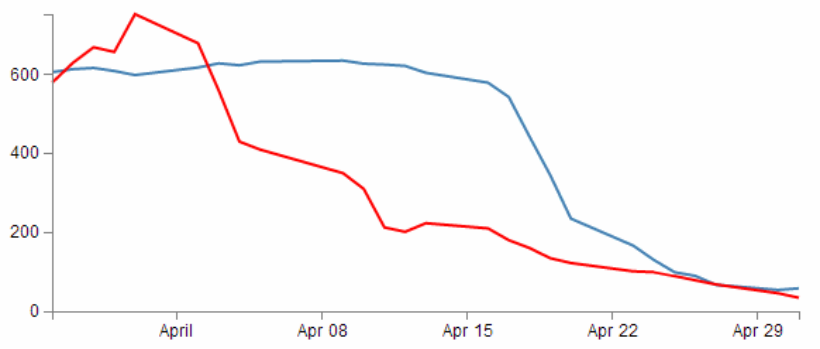
All you need to replace the `y.domain` line with is this;

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

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;

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



Two lines with everything fitting onto the canvas

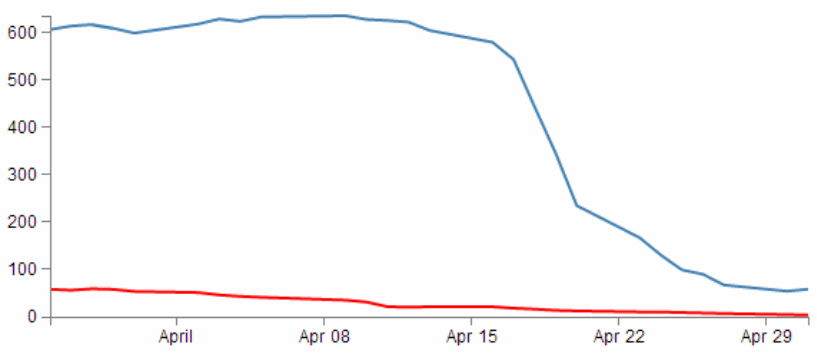
And it doesn't matter which of the two sets of data is largest, the graph will always adjust :-) You will also have noticed that our y axis has auto adjusted again to cope. Clever eh?

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 is 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[41].

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.

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.

[41] http://benjchristensen.com/2012/05/02/line-graphs-using-d3-js/

Let's get started.

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

```
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)   // This is the new declaration f\
or the 'Right', 'y1'
    .orient("right").ticks(5);                    // and includes orientation of the ax\
is to the right.
```

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); })]);
```

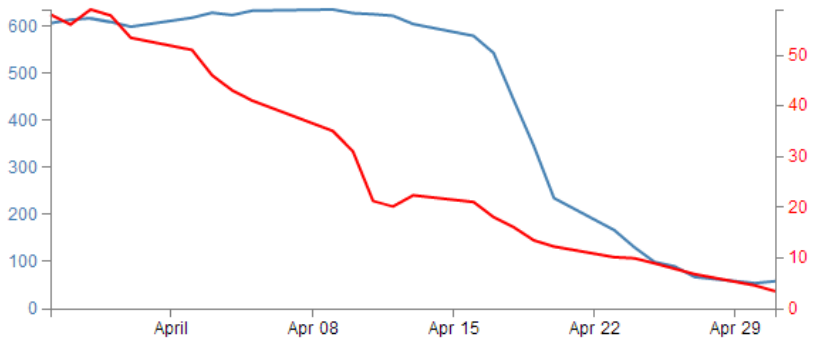
Again, here's the `y0` and `y1` changed and added and the maximums for `d.close` and `d.open` are separated out). The final piece of the puzzle is to draw the new axis, but we also want to make a slight change to the original y axis. Since we have two lines and two axes, we need to know which belongs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

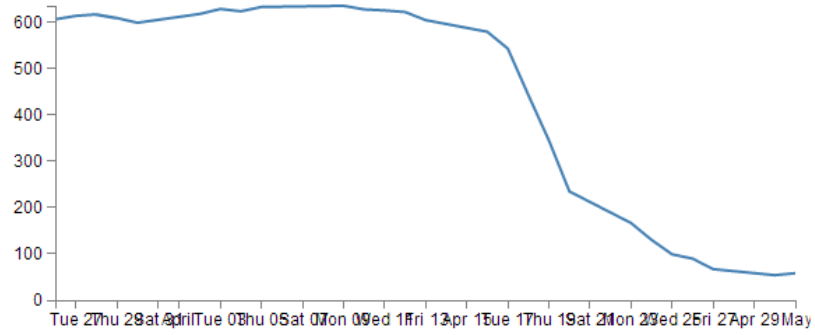


Two lines with full range of the domain and two axes

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[42].

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.

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.

[42] https://groups.google.com/forum/#!msg/d3-js/CRlW0lSbOy4/1sgrE5uS5ysJ

```
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", function(d) {
            return "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 avis 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.

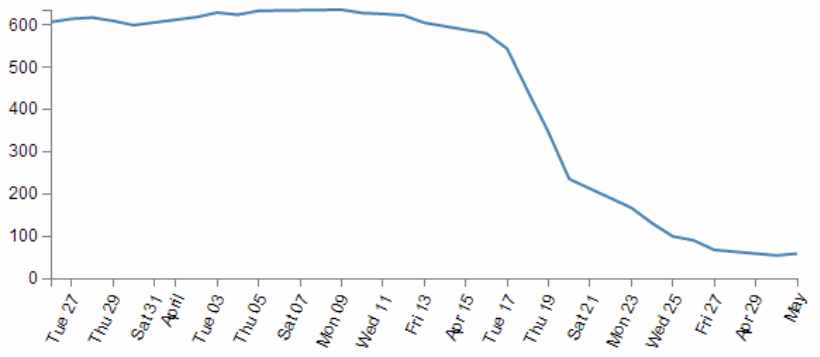
The `.style("text-anchor", "end")` line ensures that the text label has the end of the label 'attached' to the axis tick. This has the effect of making sure that the text rotates about the end of the date. This makes sure that the text all ends up at a uniform distance from the axis ticks.

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},
```

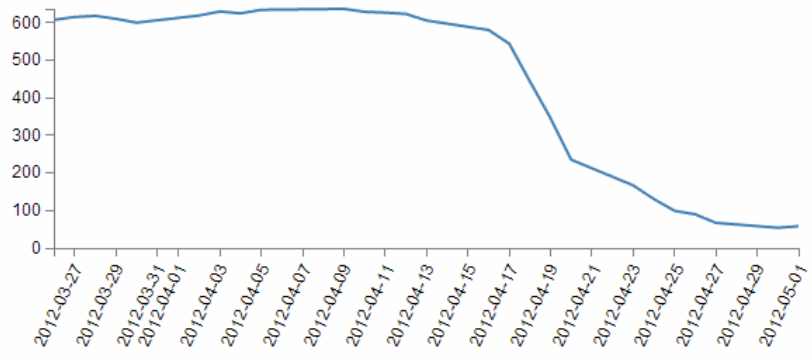
That should see us right.

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 funct\
ion
```

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](https://github.com/mbostock/d3/wiki/Time-Formatting. 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;



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)?

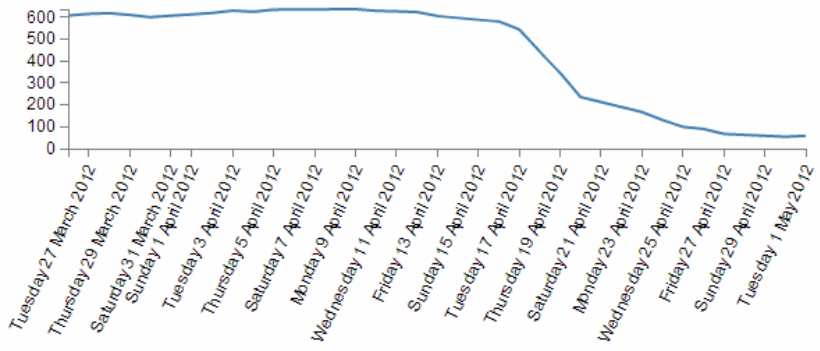
How about the full weekday name (`%A`), the day (`%d`), the full month name (`%B`) and the year (`%Y`) as a four digit number?

```
    .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[a] was the ability to provide greater control and scope to animating data.

[a] http://mbostock.github.com/d3/tutorial/protovis.html

So what are we going to do?

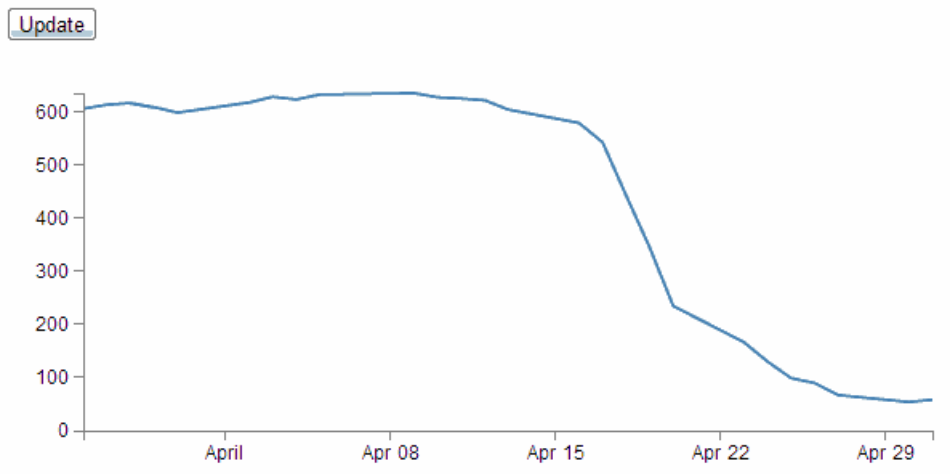
First we'll spend a bit of time setting the scene. We'll add a button to our basic graph file so that we can control when our animation occurs, we'll generate a new data set so that we can see how the data changes easily, then we'll shuffle the code about a bit to make it do it's magic. While we're shuffling 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.

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;



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 attribute (options) for allowing user input. Check out the links to w3schools[43] and Mozilla[44] for a whole lot of reading.

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

- name
- type
- value
- onclick

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

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

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.tsv` (you should be able to find it in the example file collection in the downloads page on d3noob.org). 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 oblivious 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 `</style>` tag;

```
function updateData() {

    // Get the data again
    d3.tsv("data/data-alt.tsv", 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.tsv("data/data-alt.tsv"`. 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.tsv`. 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 to 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 animation dynamic data and visualizations and is a real treasure.

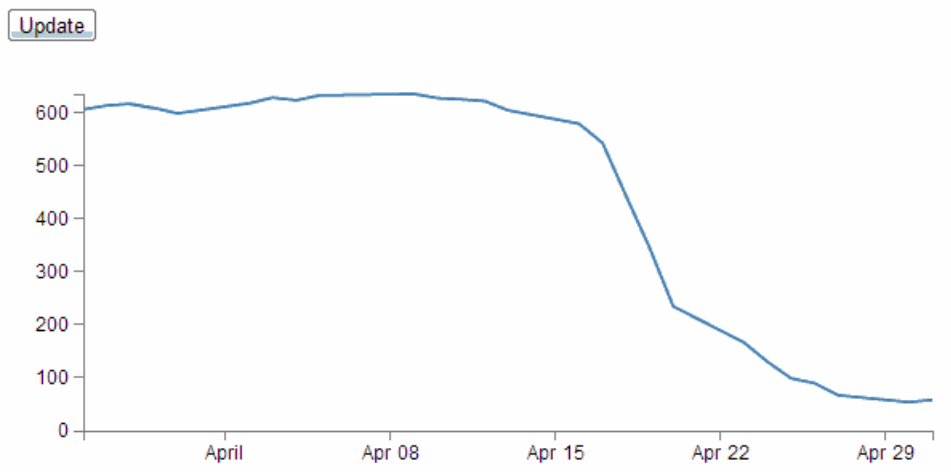
Transition Training

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).

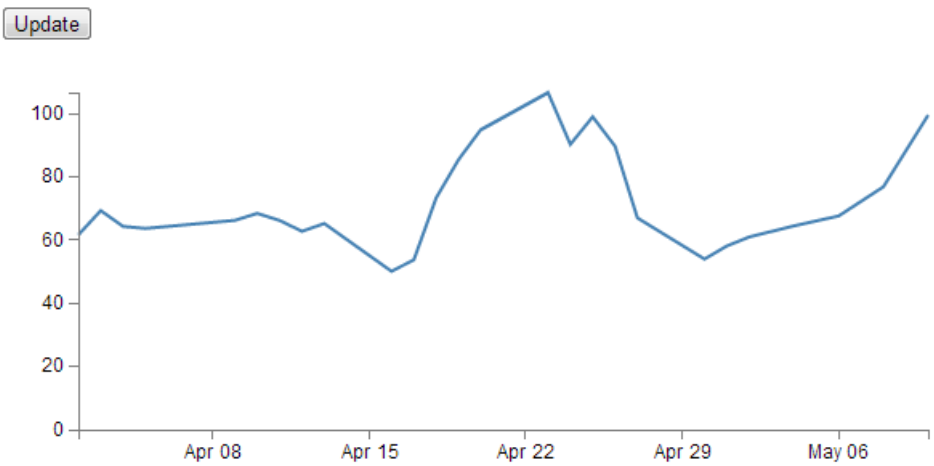
As the name suggests, a transition is a method for moving from one state to another. In it's simplest form for a d3.js visualisation, it could mean moving an object from one place to another, or changing an objects properties such as opacity or colour. In our case, we will take our data which is in the 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.tsv`) 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!

I have loaded the file for this into the d3noob downloads page with the general example files as data-load-button.html.

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 into the d3noob downloads page with the general example files as data-load-revert-button.html. There are more elegant ways to code this, but the example I give is pretty easy to follow.

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.tsv` initially, then when the button was pushed, we loaded `data-alt.tsv`. 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 in 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!

How to test?

Well, just load up your new file (I've called the one that's in the d3noob downloads page with the general example files data-load-automatic.html). 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.tsv 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.tsv file, in which case you can tell it to stop that nonsense by going into the settings and clearing the cache.

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.

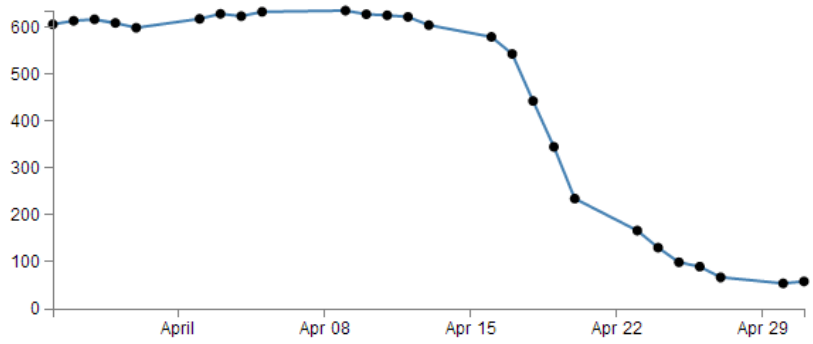
I did want to have a scatter plot, because I wanted to display tool tips, but this is too neat to ignore. It was literally a 5 minute job, 3 minutes of which was taken up by going to the d3 gallery on the wiki[a] and ogling at the cool stuff there before snapping out of it and going to the scatter plot example[b].

[a] https://github.com/mbostock/d3/wiki/Gallery
[b] http://bl.ocks.org/3887118

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.

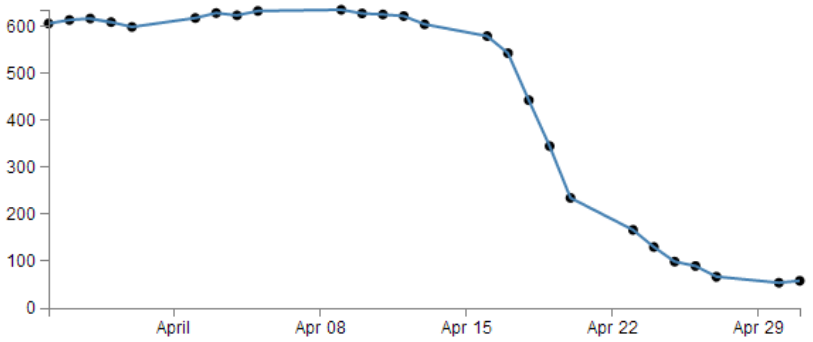
```
svg.selectAll("dot")
    .data(data)
.enter().append("circle")
    .attr("r", 3.5)
    .attr("cx", function(d) { return x(d.date); })
    .attr("cy", function(d) { return y(d.close); });
```

And you will get...



A scatter plot! (with a line)

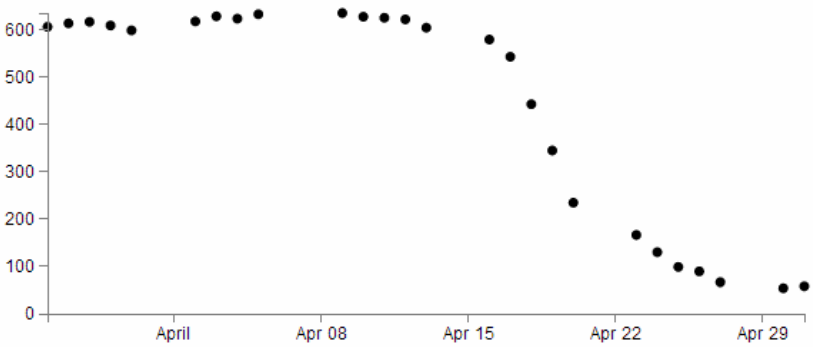
I deliberately put the dots *after* the line in the drawing section, because I thought they would look better, but you could put the block of code before the line drawing block to get the following effect;



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 looses something.

So what do the individual lines in the scatter plot block of JavaScript do?

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[45]) 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!

I've placed a copy of the file for drawing the scatter plot into the downloads section on d3noob.org with the general examples as simple-scatterplot.html.

[45] http://bl.ocks.org/3887118

Adding tooltips.

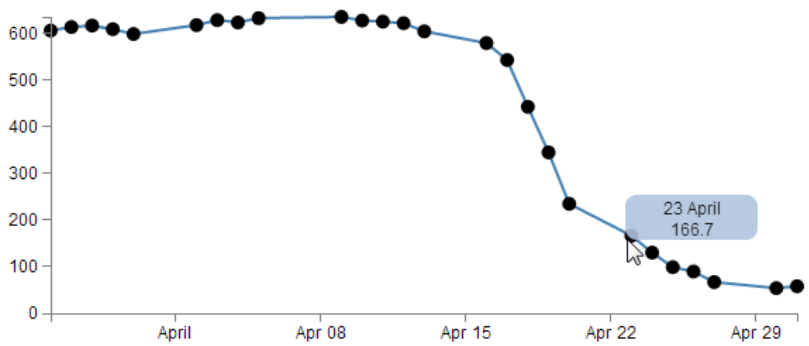
Tooltips have a marvellous duality. They are on one hand a pretty darned useful thing that aids in giving context and information where required and on the other hand, if done with a bit of care, they can look very stylish :-).

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[a]. 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[b]; (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.

[a] http://bl.ocks.org/1087001
[b] https://groups.google.com/forum/?fromgroups=#!topic/d3-js/GgFTf24ltjc

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 is hovered 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 it's 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. Purty.

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 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[46]
- Christophe Viau's 'Try D3 Now!' tutorial[47]

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[48] for a far better description of the topic if required.

[46] http://mbostock.github.com/d3/tutorial/bar-2.html
[47] http://christopheviau.com/d3_tutorial/
[48] http://javascript.info/tutorial/mouse-events

Get tipping

So, bolstered with a couple of new concepts to consider, let's see how they are enacted in practice.

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;

```
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 boarder 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[49]). That means that even if the tooltip partly obscures the circle, the code will stll 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'.

[49] https://developer.mozilla.org/en-US/docs/CSS/pointer-events

```
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!

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 to 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 it's introduction (`.transition()`). The next two lines describe the transition. It will take 200 milliseconds (`.duration(200)`) and will result in changing the elements 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 it's thing. I've placed a copy of the file for drawing the tooltips into the downloads section on d3noob.org with the general examples as tooltips.html.

What are the predefined, named colours?

Throughout this document I have been using 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.

I was really interested in what the names were for the colours. After a cursory search I was able to find a great list on about.com at http://webdesign.about.com/od/colorcharts/l/bl_namedcolors.htm[50].

The overriding point of all this is that there's more than one way to define colours in your graphs.

It means that

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

and…

```
.style("fill", "#4682b4")
```

and…

```
.style("fill", "rgb(70,130,180)")
```

All three alternatives result in the same colour being applied.

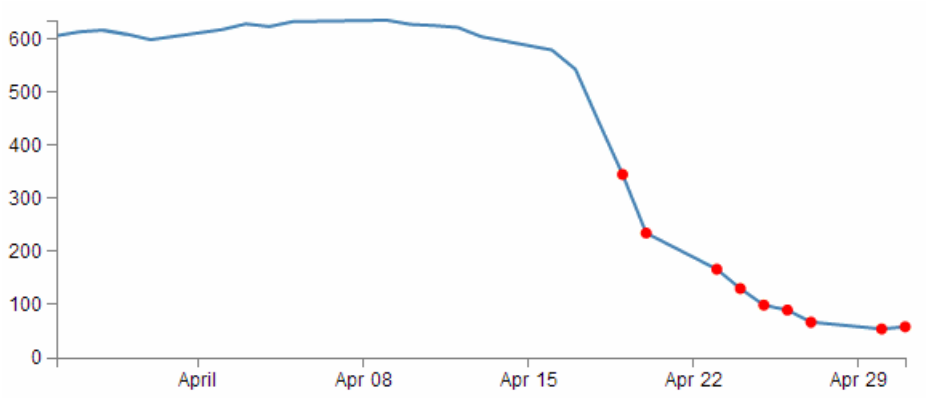
[50] http://webdesign.about.com/od/colorcharts/l/bl_namedcolors.htm

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 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`).

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;

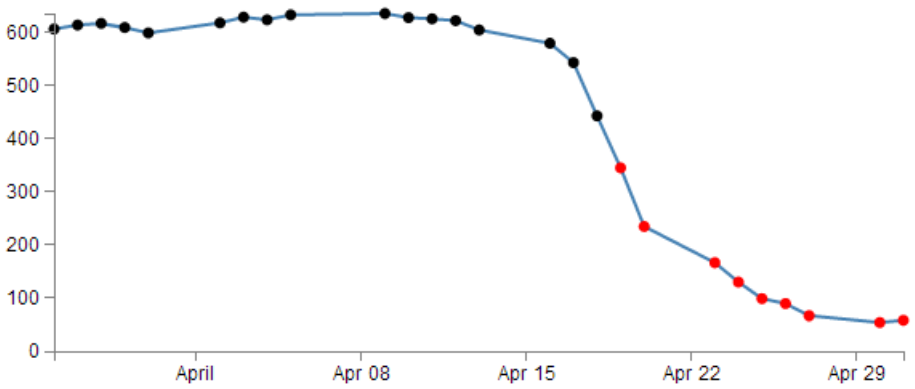
```
svg.selectAll("dot")
    .data(data)
.enter().append("circle")
    .attr("r", 3.5)
    .style("fill", function(d) {            // <== Add these
        if (d.close <= 400) {return "red"}  // <== Add these
        else         { return "black" }          // <== Add these
    ;})                                     // <== 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.

I've placed a copy of the file that uses the if statement into the downloads section on d3noob.org with the general examples as if-statement.html.

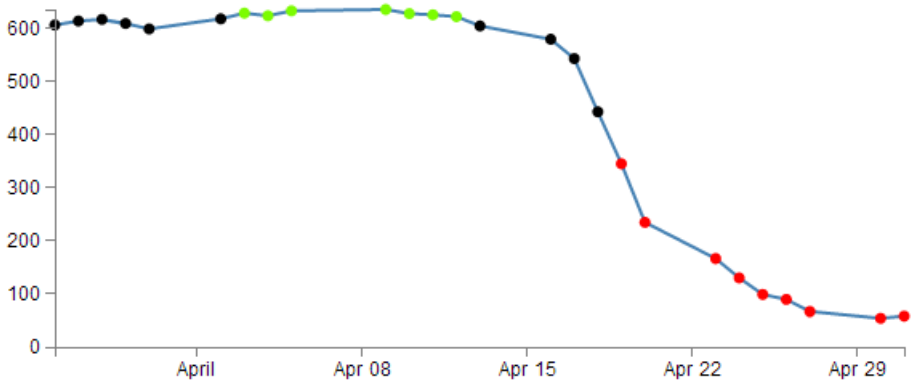
Could it be any cooler? I'm glad you asked.

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

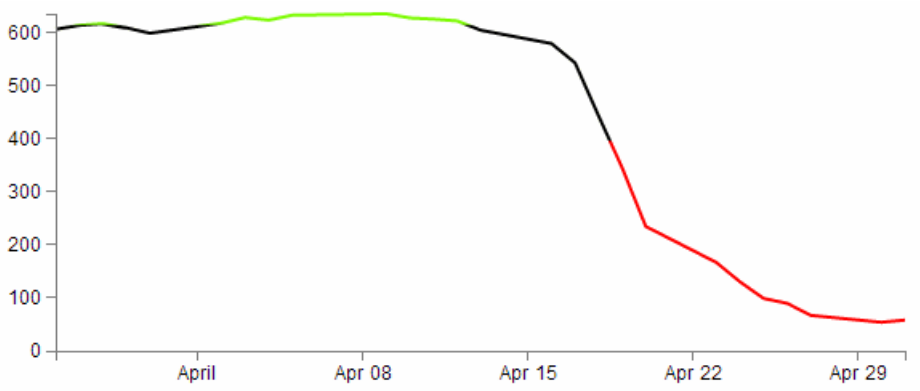
Nice.

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[51]. And I should take the opportunity to heartily recommend browsing through his collection of examples on bl.ocks.org[52]. 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.

So, first the CSS.

```
.line {
    fill: none;
    stroke: url(#line-gradient);
    stroke-width: 2px;
}
```

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.

[51] http://bl.ocks.org/3970883
[52] http://bl.ocks.org/mbostock

In this case the stroke (the colour of the line) us 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[53] (more than I ever thought existed).

The second line (`.attr("id", "line-gradient")`) sets our anchor for the CSS that we saw earlier.

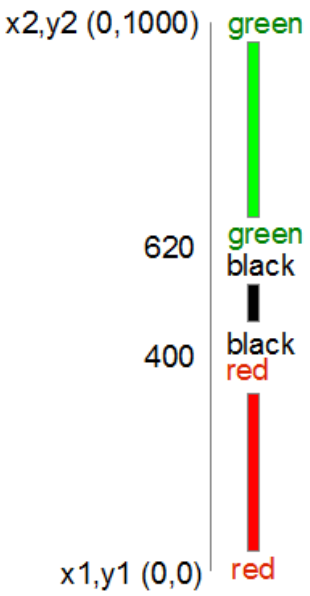
The third fourth and fifth lines define the bounds of the area over which the gradient will act. Since the coordinates `x1`, `y1`, `x2`, `y2` will describe an area. The values for `y1` (0) and `y2`

[53] http://www.w3.org/TR/SVG/pservers.html

(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[54]. We'll come back to 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% =0.43)).

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 coordinated 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 are 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 color that we defined in the stop elements area.

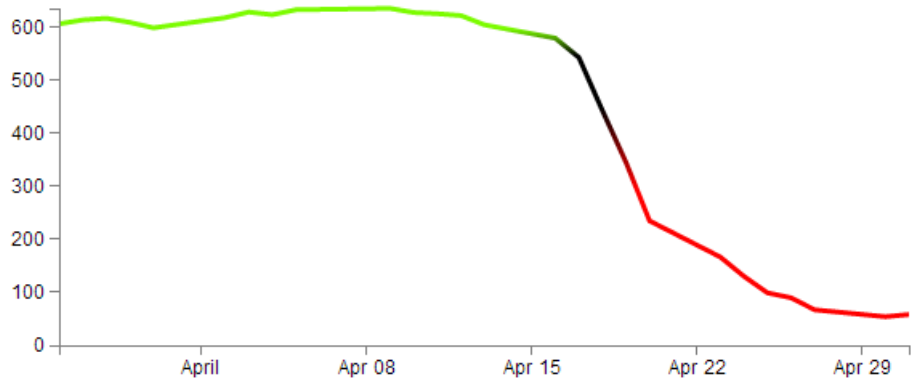
[54] https://developer.mozilla.org/en-US/docs/SVG/Attribute/gradientUnits

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")
```

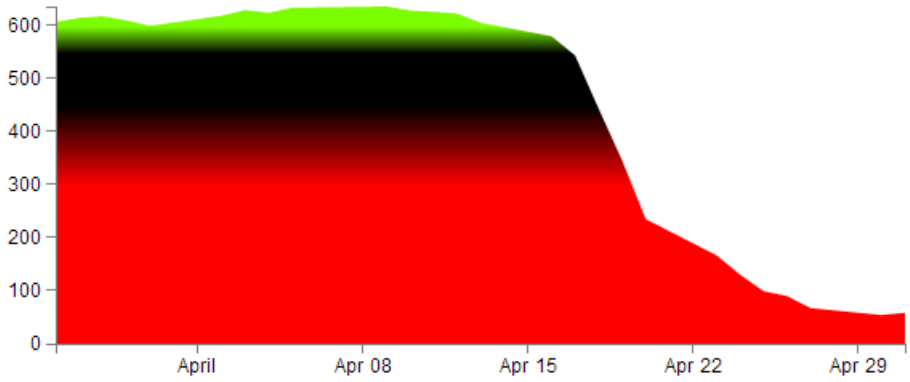
And lastly, we remove the block of code that drew the line and replace it with a block that draws an area. So change this....

```
    svg.append("path")
        .attr("class", "line")
        .attr("d", valueline(data));
```

... for 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

For a slightly 'nicer' looking example, you could check out a variation of one of Mike Bostocks originals here; http://bl.ocks.org/4433087[55].

[55] http://bl.ocks.org/4433087

Add an HTML table to your graph

So graphs and graphics are D3's bread and butter you'd think. Hmm...

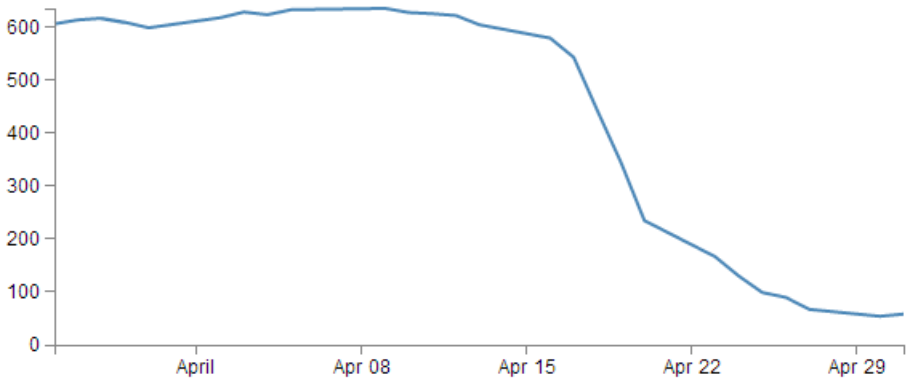
Well yes and no.

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[56] on Google Groups[57]. In fact, the post itself is an excellent one if you are considering creating a table straight from a csv file.

HTML Tables

[56] http://jsfiddle.net/7WQjr/

[57] http://stackoverflow.com/questions/9268645/d3-creating-a-table-linked-to-a-csv-file

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, 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 equates 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).

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…

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;

```
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"]);
```

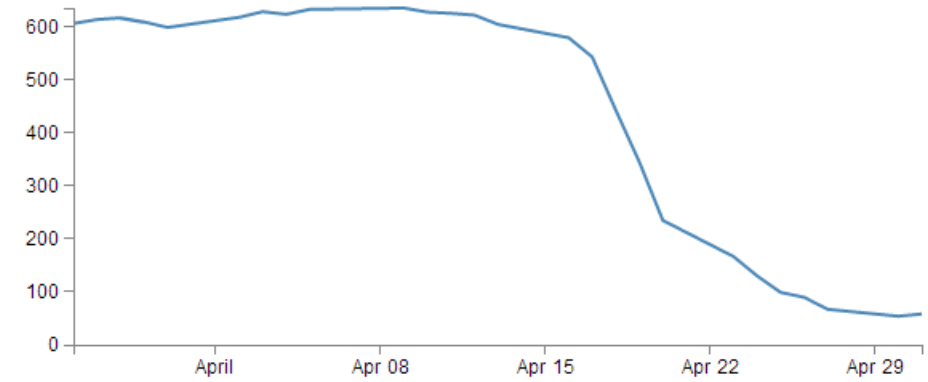
And we should take care to add it into the code at the end of the portion where we've finished drawing the graph, but before the enclosing curly and regular brackets that complete the portion of the graph that has loaded our data.tsv file. This is because we want our new piece of code to have access 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.tsv 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.tsv 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 or 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 its 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 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.

This file has been saved as table-plus-graph.html and has been added into the downloads section on d3noob.org with the general examples files.

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[58] on Google Groups[59]) and some neat additions from Christophe Viau[60] comes extra coolness that I didn't include in the previous example :-).

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 tsv file to use, so let's resurrect our old friend data2.tsv 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.tsv` to `data2.tsv` here;

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

This makes the assumption that you still have the data2.tsv file in place. If not, rush away and get it from d3noob.org's downloads page.

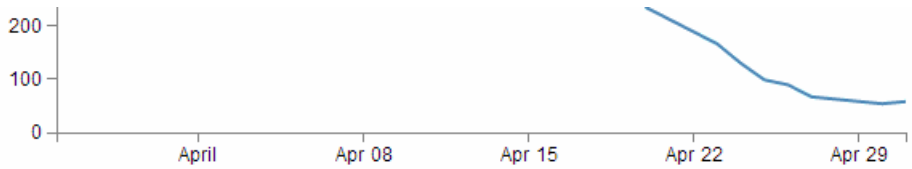
From here (and as promised in the previous chapter), it's just a matter of adding in the extra column you want (in this case it's the `open` column) like so;

```
var peopleTable = tabulate(data, ["date", "close", "open"]);
```

[58] http://jsfiddle.net/7WQjr/

[59] http://stackoverflow.com/questions/9268645/d3-creating-a-table-linked-to-a-csv-file

[60] http://christopheviau.com/d3_tutorial/



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...

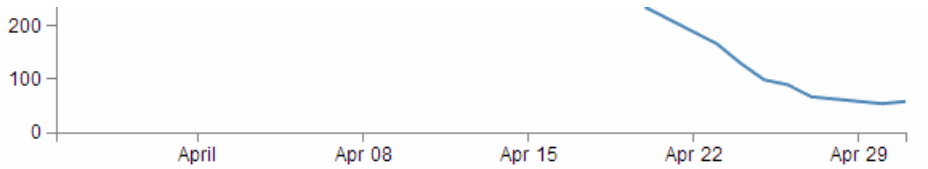
In the section where we get our data and format it, lets add another column to our array in the form of a difference between the `close` value and the `open` value (and we'll call it `diff`).

```
d3.tsv("data/data2.tsv", 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"]);
```



date	close	open	diff
1-May-12	58.13	34.12	24.01
30-Apr-12	53.98	45.56	8.42
27-Apr-12	67	67.89	-0.89
26-Apr-12	89.7	78.54	11.16
25-Apr-12	99	89.23	9.77
24-Apr-12	130.28	99.23	31.05

Table with extra extra column

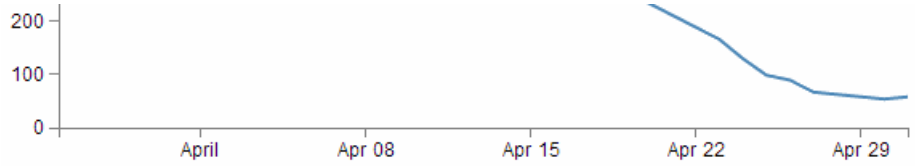
And yes, I changed the table indent again. I am a serial offender and will continue to change it to suit.

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;



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
4-Apr-12	624.31	430.56	193.75

Table sorted descending on 'close'

Prettifying (actually just capitalising the header for each column)

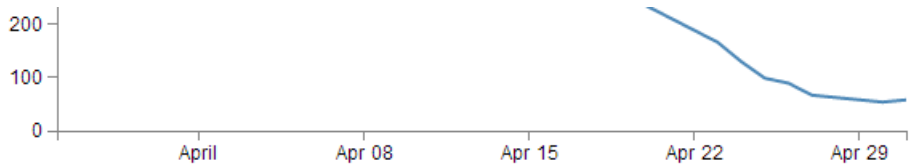
Just a little snippet that capitalises the headers for each row to make them look slightly more authoritative.

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 capitilised first characters in headers

Add borders

Sure our table looks nice and neatly arranged, but would a border look better?

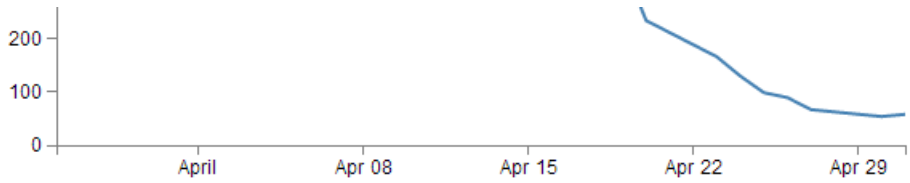
Well, here's one way to do it;

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.



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 a border

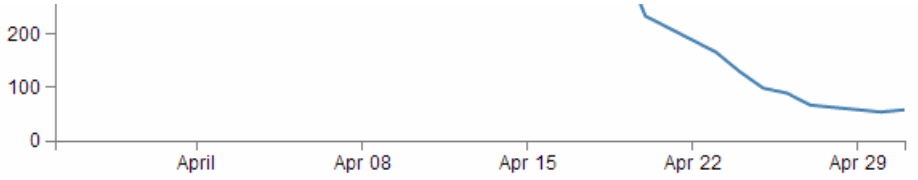
OK, so what about the individual cells?

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;



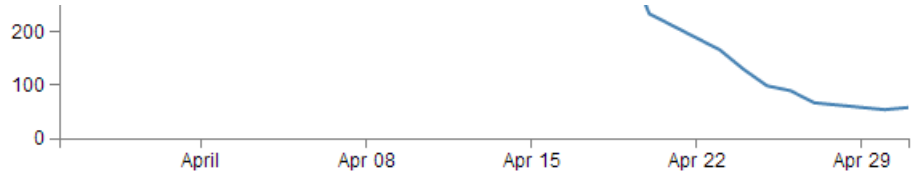
Date	Close	Open	Diff
9-Apr-12	636.23	350.45	285.78
5-Apr-12	633.68	410.23	223.45
3-Apr-12	629.32	460.34	168.98
10-Apr-12	628.44	310.45	317.99
11-Apr-12	626.2	212.67	413.53

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?



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 collapsed borders

Ahh.... Much more refined.

The `border-collapse` style tells the table to overlap each cells borders, rather than treat them as discrete entities. So in this case it looks a bit better.

This file has been saved as table-plus-addins.html and has been added into the downloads section on d3noob.org with the general examples files.

Using Plunker for development and hosting your D3 creations.

Recently Mike Bostock recommended 'Plunker' (http://plnkr.co/[61]) 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.

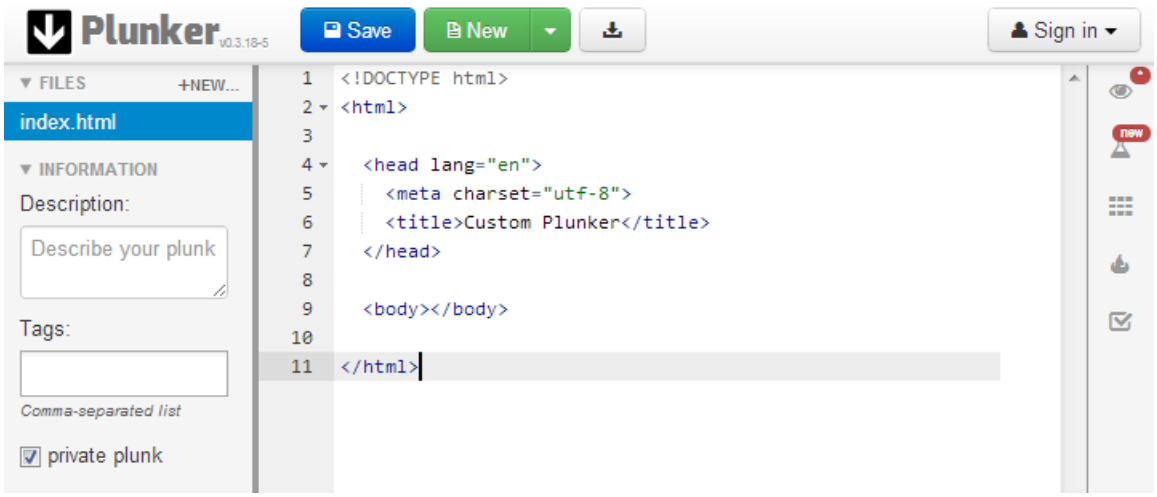
Plunker is awesome.

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/[62]



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

[61] http://plnkr.co/
[62] http://plnkr.co/edit/

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;

```
<script type="text/javascript" src="d3/d3.v3.js"></script>
```

...with this line...

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

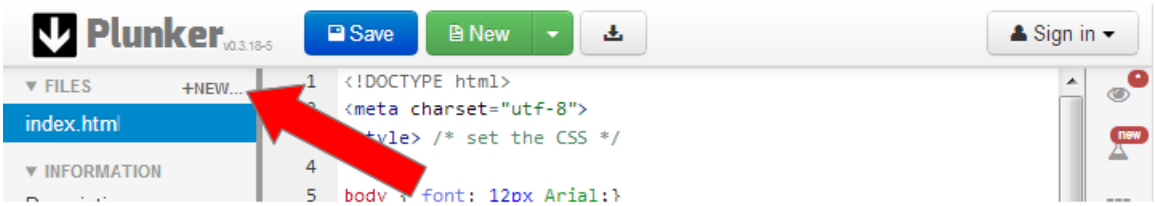
That will allow your plunk to use the version of d3.js that is hosted on d3js.org (it uses the minimised version (which is why it has the 'min' in it), but never fear, it's still d3, just distilled to enhance the flavour :-)).

Making our data available is only slightly more difficult.

In experimenting with Plunker, I found that there appears to be something 'odd' about accessing the tab separated values that we have been using thus far (in the data.tsv file), however, D3 to the rescue! We can simply use Comma Separated Values (csv) instead.

So in preparation for this exercise, please edit your data.tsv file to have the tabs separating the values replaced by commas and rename it data.csv.

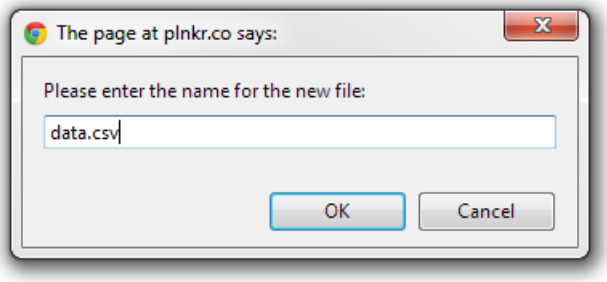
We will host our data.csv file on plunker as well and there is built in functionality to let us do it.



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 past 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

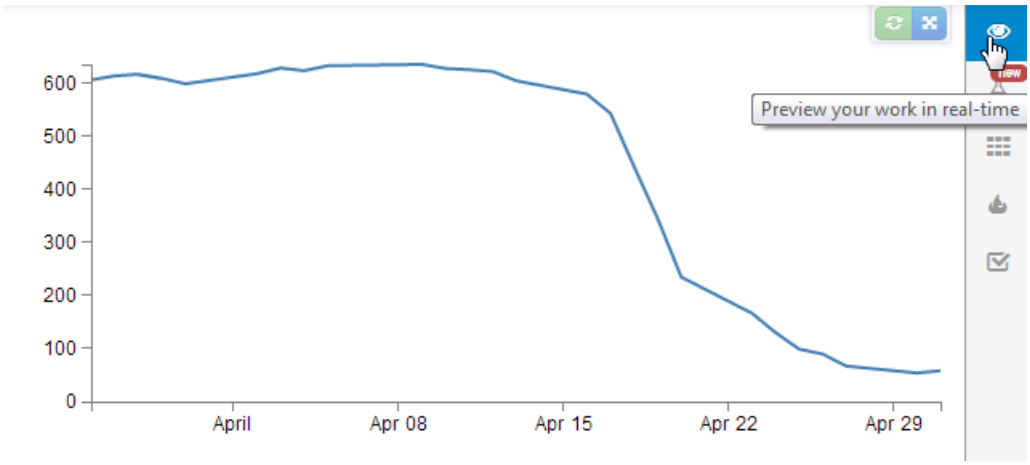
```
d3.tsv("data/data.tsv", function(error, data) {
```

... 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!



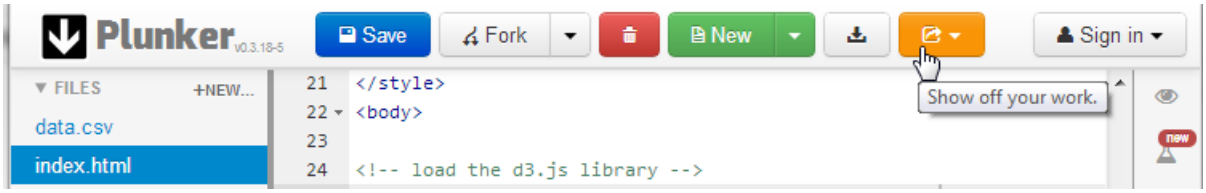
Preview your graph

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!

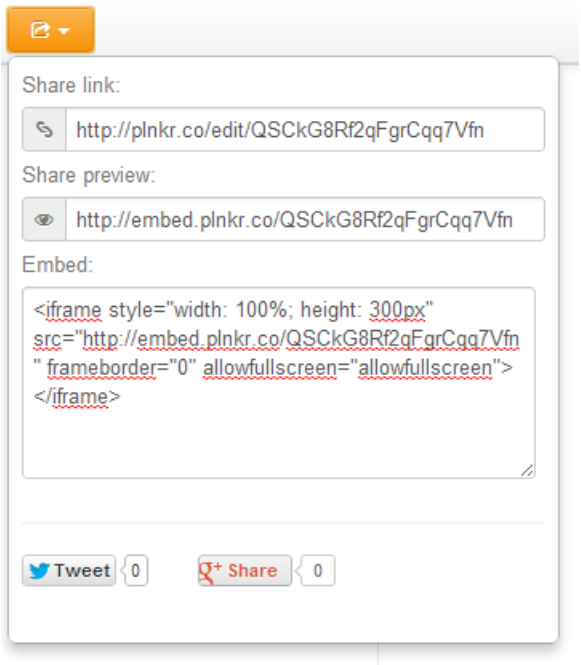
If you now click on the 'Save' button at the top of the screen, you will get some new button options.

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.



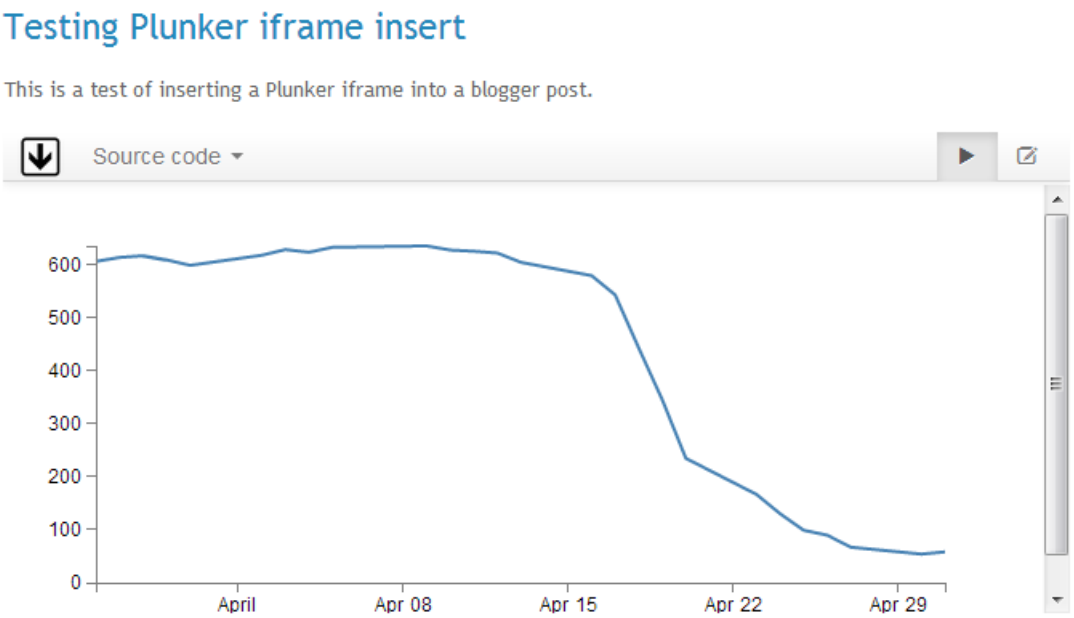
Preview your graph

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[63]

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).

[63] http://embed.plnkr.co/QSCkG8Rf2qFgrCqq7Vfn



Plunker iframe inserted in a blog post

So, I'm impressed, Nice work by Plunker and it's creator Geoff Goodman.

Using a MySQL database as a source of data.

PHP is our friend

As outlined at the start of the book, PHP is commonly used to make web content dynamic. We are going to use it to do exactly that by getting it to glue together our d3,js JavaScript and a MySQL Database. The end result should be a web page that will leverage the significant storage capability of a MySQL database and the ability to vary different aspects of returned data.

If you're wondering what level we're going to approach this at, let me reassure (or horrify) you that it will be in the same vein as the rest of this book. I am no expert in MySQL databases, but through a bit of trial and error I have been able to achieve a small measure of success. Hopefully the explanation is sufficient for beginners like myself and doesn't offend any best practices :-).

phpMyAdmin

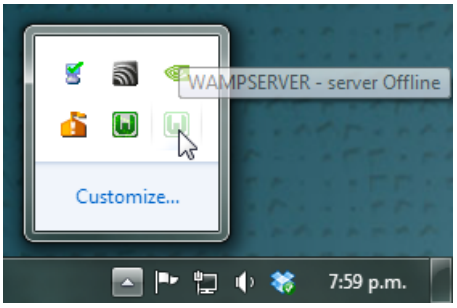
I'm not one to dwell on the command line for too long if it can be avoided (sorry). So in this section you'll see me delving into a really neat program for managing your MySQL database called phpMyAdmin (http://www.phpmyadmin.net/home_page/index.php).

As the name would suggest, it's been written in PHP and as we know, that's a sign that we're talking about a web based application. In this case phpMyAdmin is intended to allow a wide range of administrative operations with MySQL databases via a web browser. You can find a huge amount of information about it on the web as it is a freely available robust platform that has been around for well over a decade.

If you have followed my suggestion earlier in the book to install WAMP (http://www.wampserver.com/en/) or you have phpMyAdmin installed already you're in luck. If not, I'm afraid that I won't be able to provide any guidance on it's installation. I just don't have the experience to provide that level of support.

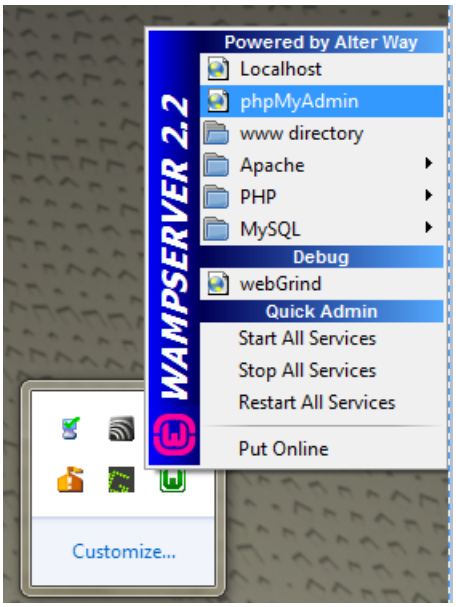
Create your database

Assuming that you do have WAMP installed, you will be able to access a subset of its functions from the icon on your system tray in the lower right hand corner of your screen.



The WAMP server icon

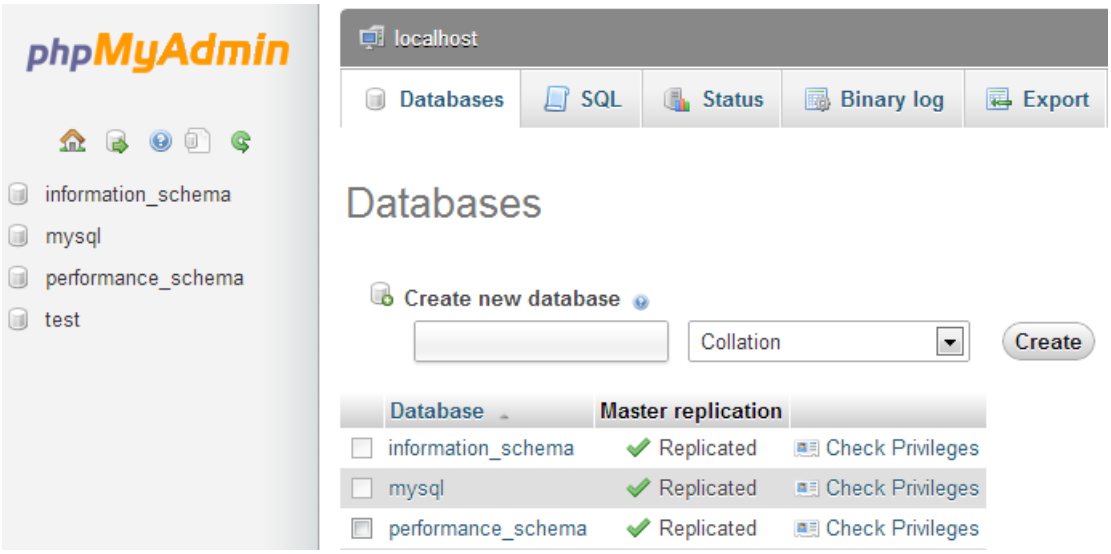
Clicking on this icon will provide you with a range of options, including opening phpMyAdmin.



Opening phpMyAdmin

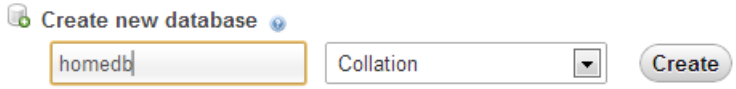
Go ahead and do this and the phpMyAdmin page will open in your browser.

The page you're presented with has a range of tabs, and we want to select the 'Databases' tab.



The Databases tab

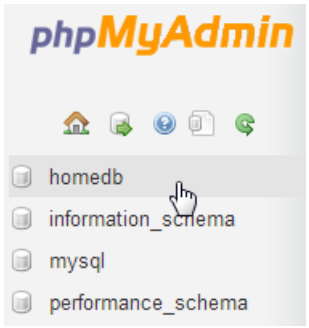
From here we can create ourselves a new database simply by giving it a name and selecting 'Create'. I will create one called 'homedb'.



Give our new database a name

That was simple!

On the panel on the left hand side of the screen is our new database. Go on and click on it.



Open the homedb database

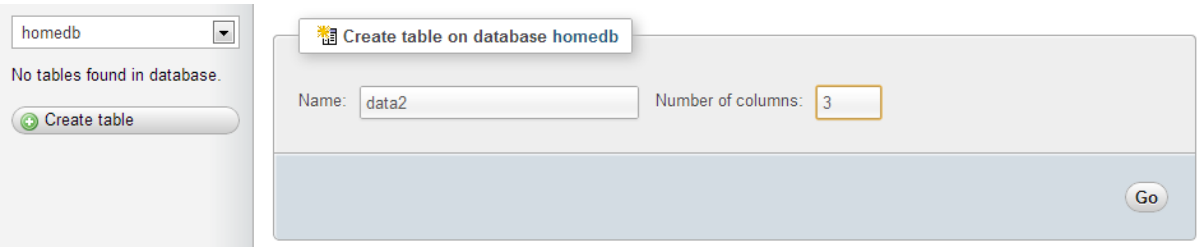
Cool, now we get to create a table. What's a table? Didn't we create our database already?

Databases and Tables

Ahh yes... Think of databases as large collections of data (yes, I can smell the

irony). Databases can have a wide range of different information stored in them, but sometimes the data isn't strictly connected. For instance, a business might want to store it's inventory and personnel records in a database. Trying to mash all that together would be a bit of a nightmare to manage. Instead, we can create two different tables of information. Think of a table as a spreadsheet with rows of data for specific columns. If we want to connect the data at some point we can do that via the process of querying the database.

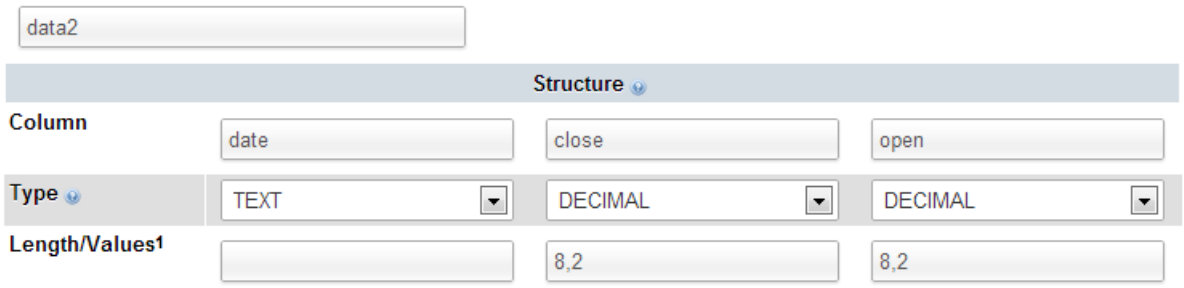
So, lets create a table called data2 with three columns.



Create a table

I've chosen data2 as a name since we will put the same data as we have in the data2.tsv file in there. That's why there are three columns for the date, close and open columns that we have in the data2.tsv file.

So, after clicking on the 'Go' button, I get the following screen where I get to enter all the pertinent details about what I will have in my table.



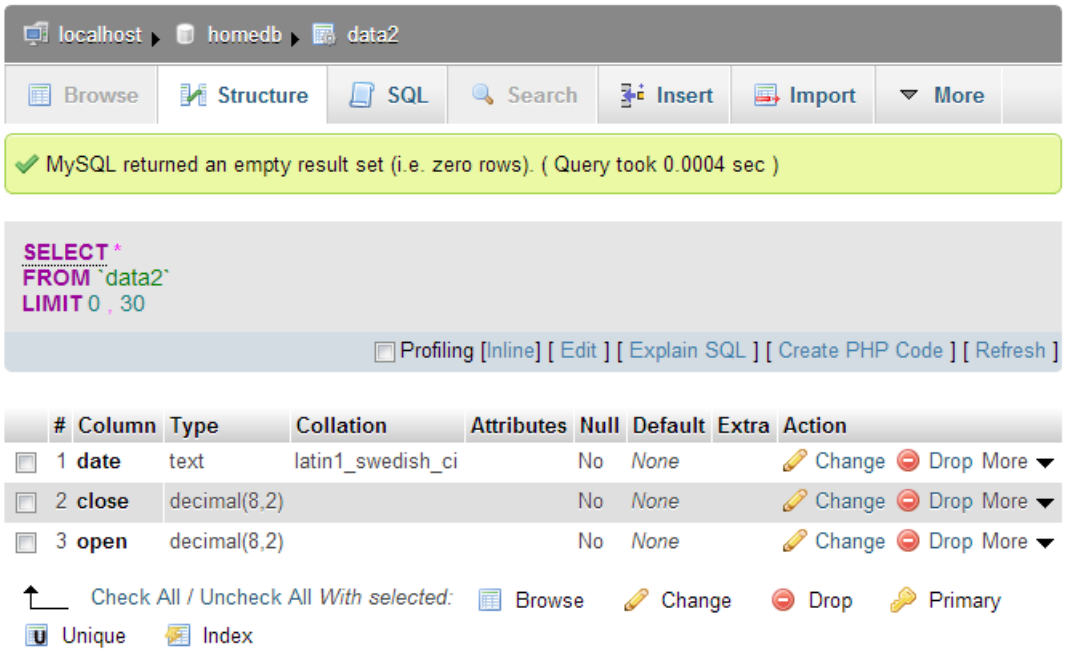
Format the table's columns

I'm keeping it really simple by setting the 'data' column to be plain text (I make the presumption that it could be a date format, but as it gets parsed into a date/time value when its ingested into D3, I'm fairly comfortable that we can get away with formatting it as 'TEXT'), and the two numeric columns to be decimals with 8 digits overall and 2 of those places for the digits to the right of the decimal point.

The selection of the most efficient data type to maximise space or speed is something of an obsession (as it sometimes needs to be) where databases are large and need to have fast access times, but in this case we're more concerned with getting a result than perfection.

Once entered, you can scroll down to the bottom of that window and select the 'Save' button.

Cool, now you are presented with your table (click on the table name in the left hand panel) and the details of it in the main panel.



The details of the 'data2' table

Sure it looks snazzy, but there's something missing..... Hmm.....

Ah Ha! Data!

Importing your data into MySQL

So, you've got a perfectly good database and an impeccably set up table looking for some data.

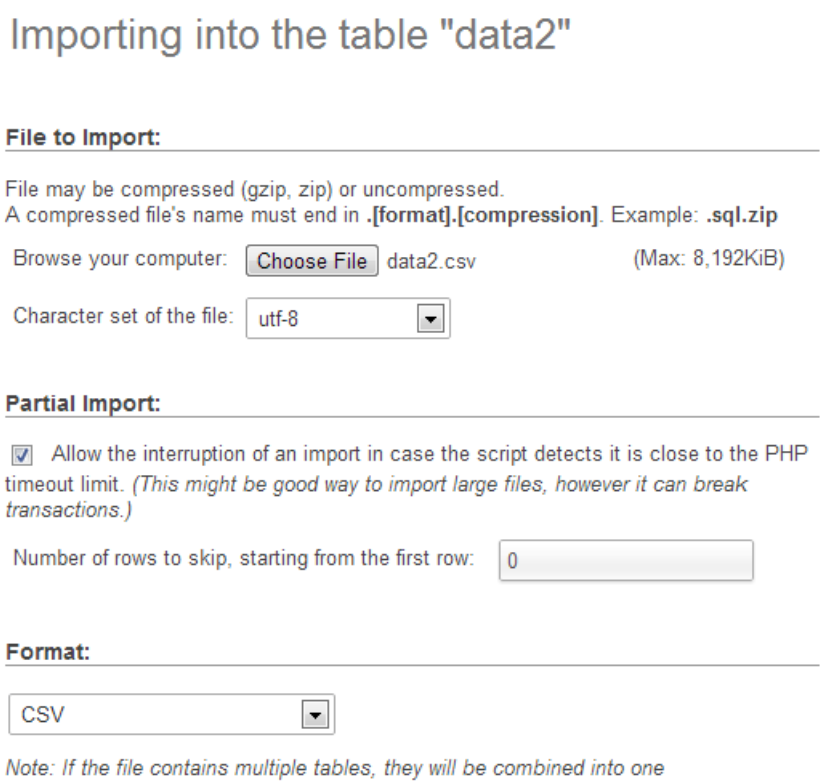
It's time we did something about that.

In the vein of "Here's one I prepared earlier", what we will do is import a csv (Comma Seperated Value) file into our database. To do this I prepared our data2.tsv file by replacing all the tabs with commas and removing the header line (with date, close and open on it), so it looks like this;

```
1-May-12,58.13,34.12
30-Apr-12,53.98,45.56
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
```

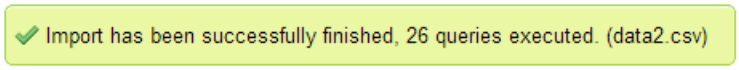
I know it doesn't look quite as pretty, but csv files are pretty ubiquitous which is why so many different programs support them as an input and output file type. (To save everyone some time and trouble I have saved the data.csv file into the D3 Tips and Tricks example files folder (under data)).

So armed with this file, click on the 'Import' tab in our phpMyAdmin window and choose your file.



Importing csv data into your table

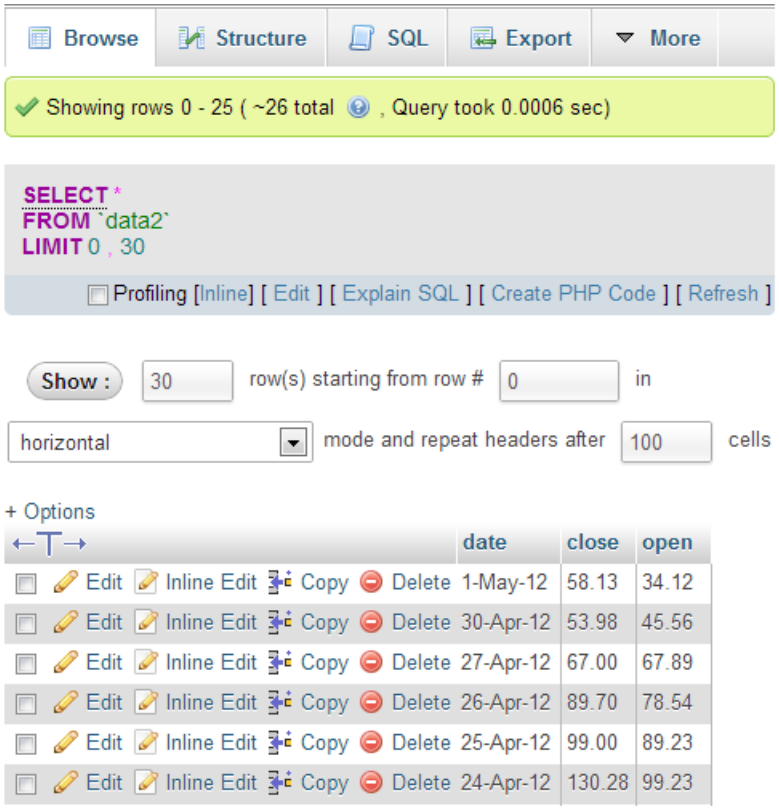
The format should be automatically recognised and the format specific options at the bottom of the window should provide sensible defaults for the input. Let's click on the 'Go' button and give it a try.



Successful import!

Woo Hoo!

Now if you click on the browse tab, there's your data in your table!



All the data successfully imported

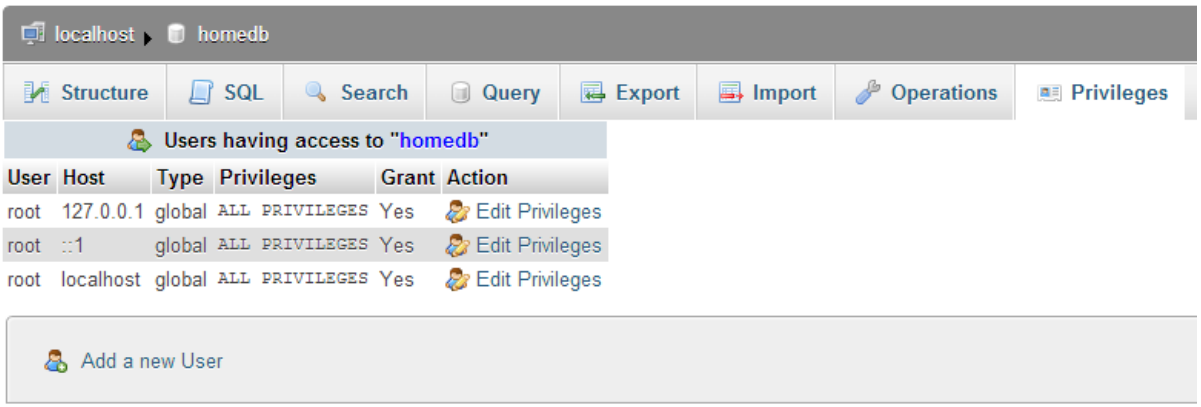
Sweet!

The last thing that we should do is add a user to our database so that we don't end up accessing it as the root user (not too much of a good look).

So select the 'homedb' reference at the top of the window (between 'localhost' and 'data2').

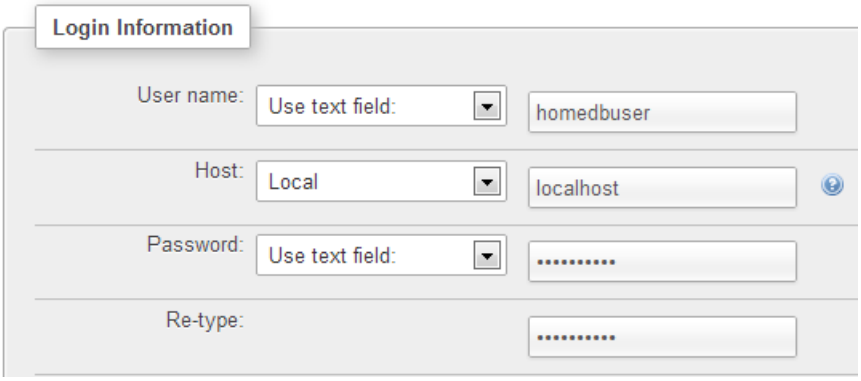
Select the 'homedb' database

Then click on the 'Privileges' tab to show all the users who have access to 'homedb' and select 'Add a new user'



The 'Privileges' tab

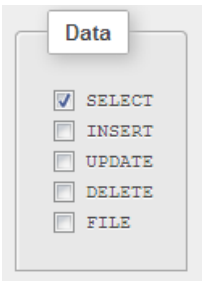
Then on the new user create a user, use the 'Local' host and put in an appropriate password.



Enter the user information

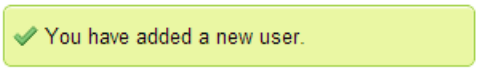
In this case, the user name is 'homedbuser' and the password is 'homedbuser' (don't tell).

The other thing to do is restrict what this untrusted user can do with the database. In this case we can fairly comfortably restrict them to 'SELECT' only;



Restrict privileges to 'SELECT'

Click on 'Go' and you have yourself a new user.



New user added!

Yay!

Believe it or not, that's pretty much it. There were a few steps involved, but they're hopefully fairly explanatory and I don't imagine there's anything too confusing that a quick Googling can't fix.

Querying the Database

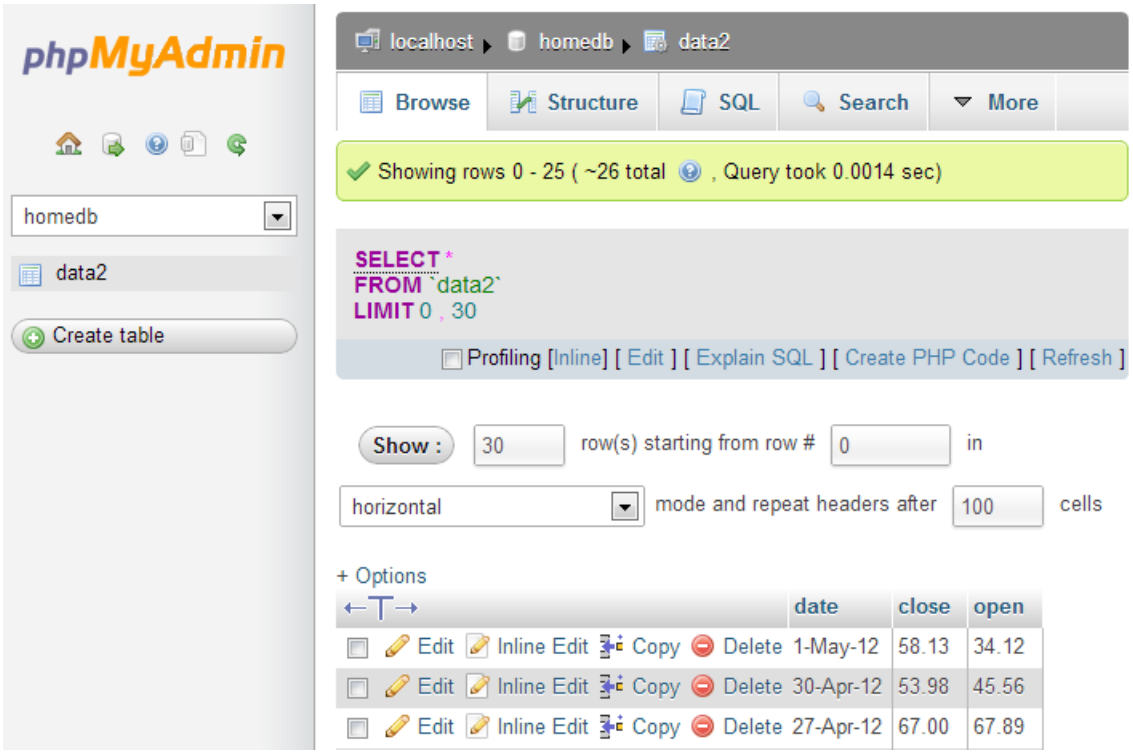
OK, are you starting to get excited yet? We're just about at the point where we can actually use our MySQL database for something useful!

To do that we have to ask the database for some information and have it return that information in a format we can work with.

The process of getting information from a database is called 'querying' the database, or performing a 'query'.

Now this is something of an art form in itself and believe me, you can dig some pretty deep holes performing queries. However, we're going to keep it simple. All we're going to do is query our database so that it returns the 'date' and the 'close' values.

We'll start by selecting our 'data2' table and going to the 'Browse' tab.



To the 'Browse' tab

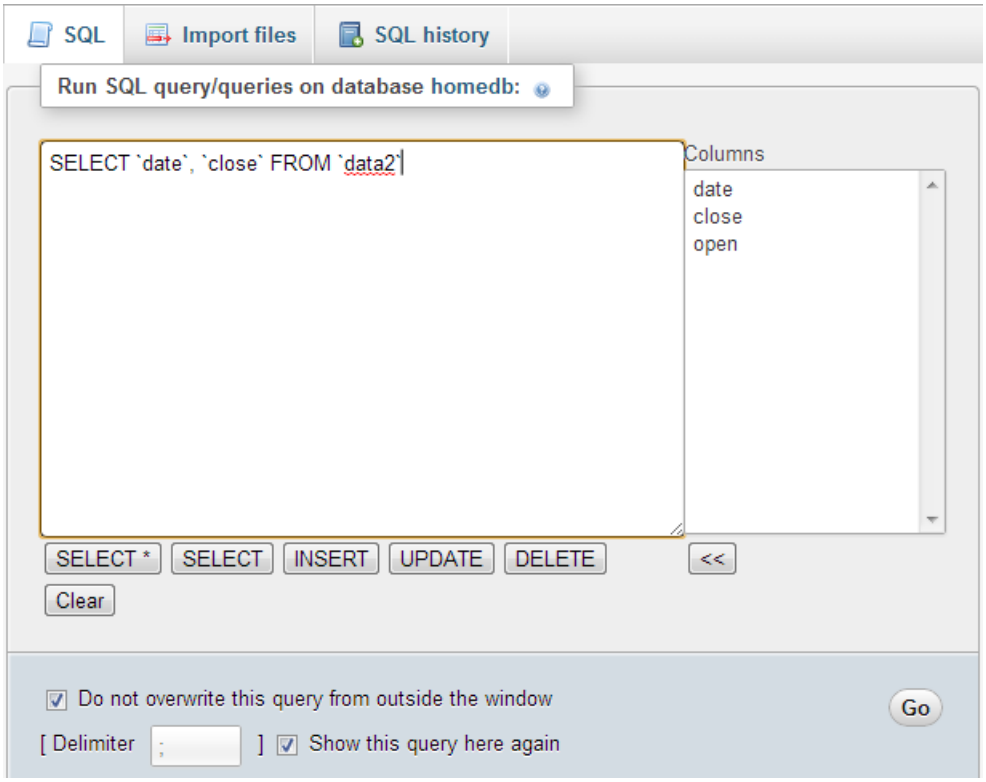
We actually already have a query operating on our table. It's the bit in the middle that looks like;

```
SELECT *
FROM `data2`
LIMIT 0, 30
```

This particular query is telling the database `homedb` (since that's where the query was run from) to `SELECT` everything (`*`) `FROM` the table `data2` and when we return the data, to `LIMIT` the returned information so those starting at record `0` and to only show `30` at a time.

You should also be able to see the data in the main body of the window.

So, let's write our own query. We can ask our query in a couple of different ways. Either click on the 'SQL' tab and you can enter it there, or click on the menu link that says 'Edit' in the current window. I prefer the 'Edit' link since it opens a separate little window which let's you look at the returned data and your query at the same time.



Enter your query

So here's our window and in it I've written the query we want to run.

```
SELECT `date`, `close` FROM `data2`
```

You will of course note that I neglected to put anything about the `LIMIT` information in there. That's because it gets added automatically to your query anyway using phpMyAdmin unless you specify values in your query.

So in this case, our query is going to `SELECT` all our values of `date` and `close` `FROM` our table `data2`.

Click on the 'Go' button and let's see what we get.

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
24-Apr-12	130.28
23-Apr-12	166.70
20-Apr-12	234.98
19-Apr-12	345.44
18-Apr-12	443.34
17-Apr-12	543.70
16-Apr-12	580.13

'date' and 'close' returned successfully

There we go!

If you're running the query as 'root' you may see lots of other editing and copying and deleting type options. Don't fiddle with them and they won't bite.

Righto... That's the query we're going to use. If you look at the returned information with a bit of a squint, you can imagine that it's in the same type of format as the *.tsv or *.csv files. (header at the top and ordered data underneath).

All that we need to do now is get out MySQL query to output data into d3.js.

Enter php!

Using php to extract json from MySQL

Now's the moment we've been waiting for to use php!

What we're going to do is use a php script that performs the query that we've just identified to extract data out of the database and to format it in a way that we can input it into D3 really easily. The data format that we're going to use for presenting to D3 is json (JavaScript Object Notation). You might remember it from the earlier chapter on types of data that could be ingested into D3.

Our php script is going to exist as a separate file which we will name `data2.php` and we will put it in a folder called `php` which will be in our webs root directory (alongside the `data` directory).

Here's the contents of our data.php file (This is repoduced in the appendices for those who prefer a stand alone version);

```
<?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);
?>
```

It's pretty short, but it packs a punch. Let's go through it and see what it does.

The `<?php` line at the start and the `?>` line at the end form the wrappers that allow the requesting page to recognise the contents as php and to execute the code rather than downloading it for display.

The following lines set up a range of important variables;

```
    $username = "homedbuser";
    $password = "homedbuser";
    $host = "localhost";
    $database="homedb";
```

Hopefully you will recognise that these are the configuration details for the MySQL database that we set up. There's the user and his password (don't worry, because the script isn't returned to the browser, the browser doesn't get to see the password and in this case our user has a very limited set of privileges remember). There's the host location of our database (in this case it's

local, but if it was on a remote server, we would just include it's address) and there's the database we're going to access.

Then we use those variables to connect to the server...

```
$server = mysql_connect($host, $username, $password);
```

... and then we connect to the specific database;

```
 $connection = mysql_select_db($database, $server);
```

Then we have our query in a form that we can paste into the right spot and it's easy to use.

```
    $myquery = "
SELECT  `date`, `close` FROM  `data2`
";
```

I have it like this so all I need to do to change the query I use is paste it into the middle line there between the speech-marks and I'm done. It's just a convenience thing.

The query is then run against the database with the following command;

```
    $query = mysql_query($myquery);
```

... and then we check to see if it was successful. If it wasn't, we output the MySQL error code;

```
    if ( ! $myquery ) {
        echo mysql_error();
        die;
    }
```

Then we declare the `$data` variable as an array (`$data = array();`) and feed the returned information from our query into `$data` array;

```
    for ($x = 0; $x < mysql_num_rows($query); $x++) {
        $data[] = mysql_fetch_assoc($query);
    }
```

(that's a fancy little piece of code that gets the information row by row and puts it into the array)

We then return (`echo`) the `$data` array in json format (`echo json_encode($data);`) into whatever ran the `data2.php` script (we'll come back to this in a minute).

Then finally we close the connection to the server;

```
    mysql_close($server);
```

Whew!

That was a little fast and furious, but I want to revisit the point that we covered in the part about echoing the data back to whatever had requested it. This is because we are going to use it directly in our d3.js script, but we can actually run the script directly but opening the file in our browser.

So if you can navigate using your browser to this file and run it (WAMP should be your friend here again) this is what you should see printed out on your screen;

```
[{"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"},
{"date":"24-Apr-12","close":"130.28"},
{"date":"23-Apr-12","close":"166.70"},
{"date":"20-Apr-12","close":"234.98"},
{"date":"19-Apr-12","close":"345.44"},
{"date":"18-Apr-12","close":"443.34"},
{"date":"17-Apr-12","close":"543.70"},
{"date":"16-Apr-12","close":"580.13"},
{"date":"13-Apr-12","close":"605.23"},
{"date":"12-Apr-12","close":"622.77"},
{"date":"11-Apr-12","close":"626.20"},
{"date":"10-Apr-12","close":"628.44"},
{"date":"9-Apr-12","close":"636.23"},
{"date":"5-Apr-12","close":"633.68"},
{"date":"4-Apr-12","close":"624.31"},
{"date":"3-Apr-12","close":"629.32"},
{"date":"2-Apr-12","close":"618.63"},
{"date":"30-Mar-12","close":"599.55"},
{"date":"29-Mar-12","close":"609.86"},
{"date":"28-Mar-12","close":"617.62"},
{"date":"27-Mar-12","close":"614.48"},
{"date":"26-Mar-12","close":"606.98"}]
```

There it is! The data we want formatted as json!

It looks a bit messy on the printed page, but it's bread and butter for JavaScript.

I have included the data2.php file in the examples zip file that can be downloaded from d3noob.org.

Getting the data into d3.js

Let's recap momentarily.

We have created a database, populated it with information, worked out how ro extract a subset of that information and how to do it in a format that d3.js understands. Now for the final act! And you will find it slightly deflating how simple it is.

All we have to do is take our simple-graph.html file and make the following change;

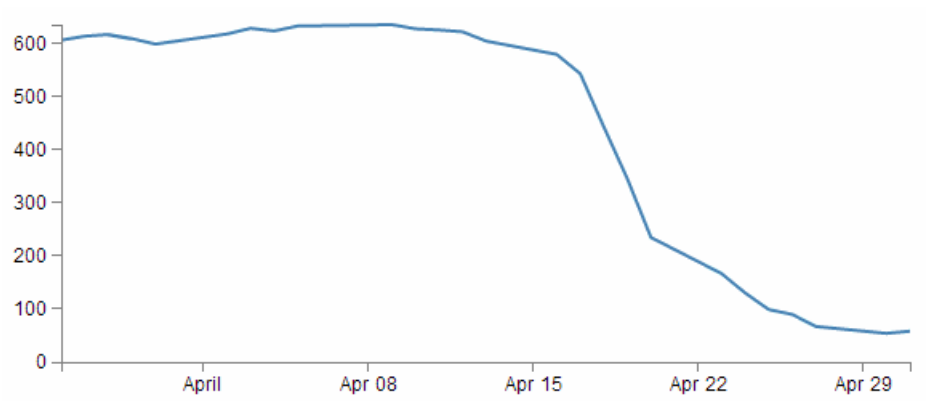
```
d3.json("php/data2.php", function(error, data) {
        data.forEach(function(d) {
                d.date = parseDate(d.date);
                d.close = +d.close;
        });
```

Here we have replaced the part of the code that read in the data file as `data.tsv` with the equivalent that reads the `php/data2.php` file in as json (`d3.json`).

That's it.

What it does is we tell d3.js to go and get a json file and when it strikes the `data2.php` file, it executes the script in the file and returns the encoded json information directly to d3.js. How cool is that?

And here is the result.



Our faithful simple line graph

Sure, it looks kind of familiar, but it represents a significant ability for you to return data from a database and present it on a web page.

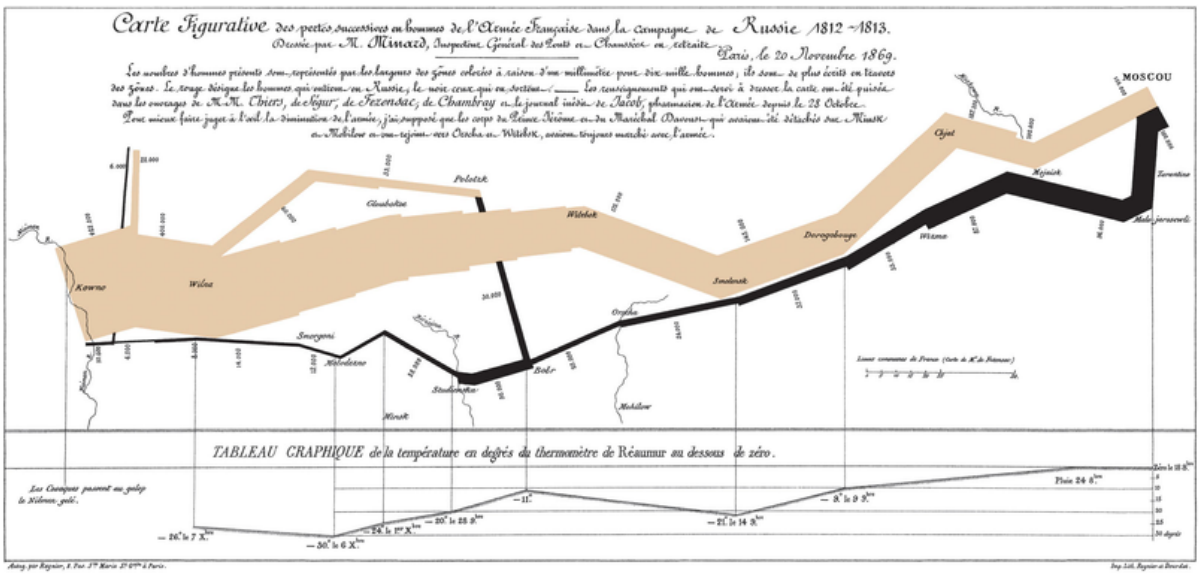
Sankey Diagrams

What is a Sankey Diagram?

A Sankey diagram is a type of flow diagram where the 'flow' is represented by arrows of varying thickness depending on the quantity of flow.

They are often used to visualize energy, material or cost transfers and are especially useful in demonstrating proportionality to a flow where different parts of the diagram represent different quantities in a system.

Probably the most famous example of a Sankey diagram is Charles Minard's Map of Napoleon's Russian Campaign of 1812.



Napoleon's Russian March

From Wikipedia;

"*Étienne-Jules Marey first called notice to this dramatic depiction of the fate of Napoleon's army in the Russian campaign, saying it defies the pen of the historian in its brutal eloquence. Edward Tufte says it "may well be the best statistical graphic ever drawn" and uses it as a prime example in The Visual Display of Quantitative Information.*"

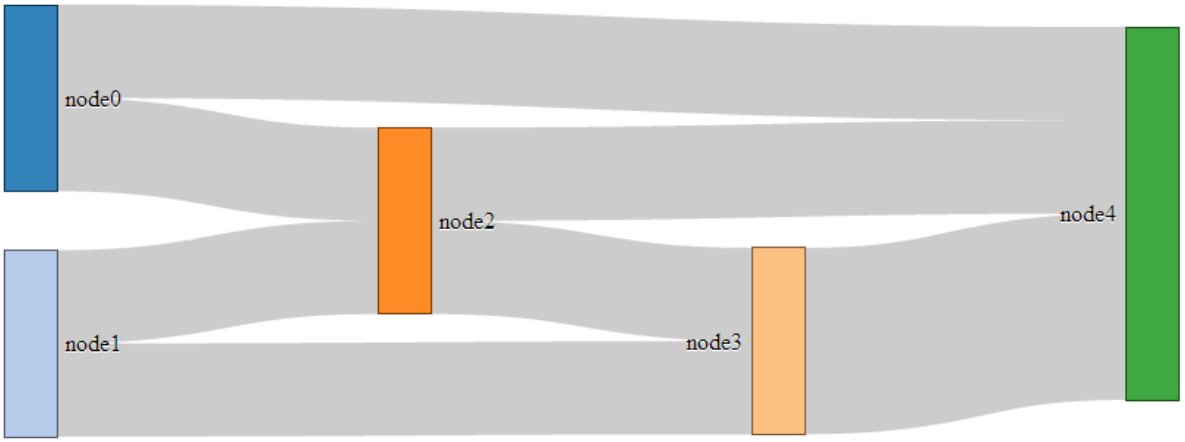
Wikipedia has a great explanation of the diagram type[64] and there is a wealth of information dedicated it on the inter-web. I heartily recommend http://www.sankey-diagrams.com/ for all things Sankey!

So it would come as little surprise that Mike Bostock has developed a plugin for Sankey diagrams (http://bost.ocks.org/mike/sankey/) so that we can all enjoy Sankey goodness with lashings of D3.

[64] http://en.wikipedia.org/wiki/Sankey_diagram

How d3.js Sankey Diagrams want their data formatted

If we think of Sankey diagrams consisting of 'nodes' and 'links'...



A simple Sankey diagram

... the data that generates them must be formatted as nodes and links as well.

For instance a JSON file with appropriate data to build the diagram above could look like the following;

```
{
"nodes":[
{"node":0,"name":"node0"},
{"node":1,"name":"node1"},
{"node":2,"name":"node2"},
{"node":3,"name":"node3"},
{"node":4,"name":"node4"}
],
"links":[
{"source":0,"target":2,"value":2},
{"source":1,"target":2,"value":2},
{"source":1,"target":3,"value":2},
{"source":0,"target":4,"value":2},
{"source":2,"target":3,"value":2},
{"source":2,"target":4,"value":2},
{"source":3,"target":4,"value":4}
]}
```

In the file above we have 6 nodes (0-5) sequentially numbered and with names appropriate to their position in the list.

The sequential numbering is only for the purpose of highlighting the structure of the data, since when we get D3 running, it will automatically index each of the nodes according to its position. In other words, we could have omitted the "node":n parts since D3 will know where each node is anyway. The big deal is that WE need to know what each node is as well especially if we're going to be building the data by hand (doing it dynamically would be cool, but let's not get ahead of ourselves just yet).

The links part of the data can be broken down into individual source to target 'links' that have an associated value (could be a quantity or strength, but at least a numeric value).

The 'source' and target numbers are references to the list of nodes. So, "source":1, "target":2 means that this link is whatever node appears at position 1 going to whatever node appears at position 2. The important point to make here is that D3 will not be interested in the numerical value of the node, just it's position in the list (starting at zero).

Description of the code

The code for the Sankey diagram is significantly different to that for a line graph although it shares the same core language and programming methodology.

The code we'll go through is an adaptation of the version first demonstrated by Mike Bostock[65] so it's got a pretty good pedigree. I will begin with a version that uses data that is formatted so that it can be used directly with no manipulation, then in subsequent sections I will describe different techniques for getting data from different formats to work.

I found that getting data in the correct format was the biggest hurdle for getting a Sankey diagram to work. I make the assumption that this may be a similar story for others as well. We will start off assuming that the data is perfectly formatted, then where only the link data is available then where there is just names to work with (no numeric node values) and lastly, one that can be used for people with changeable data from a MySQL database.

I won't try to go over every inch of the code as I did with the previous simple graph example (I'll skip things like the HTML header) and will focus on the style sheet (CSS) portion and the JavaScript.

The complete code for this will also be available as an appendix and in the downloads section at d3noob.org.

On to the code…

```
<style>
```

[65] http://bost.ocks.org/mike/sankey/

```
.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);
  }
});
```

So, going straight to the style sheet bounded by the `<style>` tags;

```
.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;
}
```

The CSS in this example is mainly concerned with formatting of the mouse cursor as it moves around the diagram.

The first part...

```
.node rect {
  cursor: move;
  fill-opacity: .9;
  shape-rendering: crispEdges;
}
```

... provides the properties for the node rectangles. It changes the icon for the cursor when it moves over the rectangle to one that looks like it will move the rectangle (there is a range of different icons that can be defined here http://www.echoecho.com/csscursors.htm), sets the fill colour to mostly opaque and keeps the edges sharp.

The next block...

```
.node text {
  pointer-events: none;
  text-shadow: 0 1px 0 #fff;
}
```

... sets the properties for the text at each node. The mouse is told to essentially ignore the text in favour of anything that's under it (in the case of moving or highlighting something else) and a slight shadow is applied for readability).

The following block...

```
.link {
  fill: none;
  stroke: #000;
  stroke-opacity: .2;
}
```

... makes sure that the link has no fill (it actually appears to be a bendy rectangle with very thick edges that make the element appear to be a solid block), colours the edges black (`#000`) and gives makes the edges almost transparent.

The last block....

```
.link:hover {
  stroke-opacity: .5;
}
```

... simply changes the opacity of the link when the mouse goes over it so that it's more visible. If so desired, we could change the colour of the highlighted link by adding in a line to this block changing the colour like this `stroke: red;`.

Just before we get into the JavaScript, we do something a little different for d3.js. We tells it to use a plug-in with the followig line;

```
<script src="js/sankey.js"></script>
```

The concept of a plug-in is that it is a separate piece of code that will allow additional functionality to a core block (which in this case is d3.js). There are a range of plug-ins available[66] and we will need to source the `sankey.js` file from the repository and place that somewhere where our HTML code can access it. In this case I have put it in the `js` directory that resides in the root directory of the web page.

The start of our JavaScript begins by defining a range of variables that we'll be using.

Our units are set as 'Widgets' (`var units = "Widgets";`), which is just a convenient generic (nonsense) term to provide the impression that the flow of items in this case is widgets being passed from one person to another.

We then set our canvas size and margins...

```
var margin = {top: 10, right: 10, bottom: 10, left: 10},
    width = 700 - margin.left - margin.right,
    height = 300 - margin.top - margin.bottom;
```

... before setting some formatting.

[66] https://github.com/d3/d3-plugins

```
var formatNumber = d3.format(",.0f"),    // decimal places
    format = function(d) { return formatNumber(d) + " " + units; },
    color = d3.scale.category20();
```

The `formatNumber` function acts on a number to set it to zero decimal places in this case. In the original Mike Bostock example it was to three places, but for 'widgets' I'm presuming we don't divide :-).

`format` is a function that returns a given number formatted with `formatNumber` as well as a space and our units of choice ('Widgets'). This is used to display the values for the links and nodes later in the script.

The `color = d3.scale.category20();` line is really interesting and provides access to a colour scale that is pre-defined for your convenience[67]!. Later in the code we will see it in action.

Our next block of code positions our canvas onto our page in relation to the size and margins we have already defined;

```
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 + ")");
```

Then we set the variables for our sankey diagram;

```
var sankey = d3.sankey()
    .nodeWidth(36)
    .nodePadding(40)
    .size([width, height]);
```

Without trying to state the obvious, this sets the width of the nodes (`.nodeWidth(36)`), the padding between the nodes (`.nodePadding(40)`) and the size of the diagram(`.size([width, height]);`).

The following line defines the `path` variable as a pointer to the sankey function that make the links between the nodes to their clever thing of bending into the right places.;

```
var path = sankey.link();
```

I make the presumption that this is a defined function within `sankey.js`. Then we load the data for our sankey diagram with the following line;

[67] https://github.com/mbostock/d3/wiki/Ordinal-Scales#wiki-category10

```
d3.json("data/sankey-formatted.json", function(error, graph) {
```

As we have seen in previous usage of the `d3.json`, `d3.csv` and `d3.tsv` functions this is a wrapper that acts on all the code within it bringing the data in the form of `graph` to the remaining code.

I think it's a good time to take a slightly closer look at the data that we'll be using;

```
{
"nodes":[
{"node":0,"name":"node0"},
{"node":1,"name":"node1"},
{"node":2,"name":"node2"},
{"node":3,"name":"node3"},
{"node":4,"name":"node4"}
],
"links":[
{"source":0,"target":2,"value":2},
{"source":1,"target":2,"value":2},
{"source":1,"target":3,"value":2},
{"source":0,"target":4,"value":2},
{"source":2,"target":3,"value":2},
{"source":2,"target":4,"value":2},
{"source":3,"target":4,"value":4}
]}
```

I want to look at the data now, because it highlights how it is accessed throughout this portion of the code. It is split into two different blocks, 'nodes' and 'links'. The subset of variables available under 'nodes' is 'node' and 'name'. Likewise under 'links' we have 'source', 'target' and 'value'. This means that when we want to act on a subset of our data we define which piece by defining the hierarchy that leads to it. For instance, if we want to define an action for all the links, we would use graph.links (they're kind of chained together).

Let me take this opportunity to apologise to all those programmers who *actually* know exactly what is going on here. It's a mystery to me, but this is how I like to tell myself it works to help me get by :-).

Now that we have our data loaded, we can assign the data to the sankey function so that it knows how to deal with it behind the scenes;

```
sankey
    .nodes(graph.nodes)
    .links(graph.links)
    .layout(32);
```

In keeping with our previous description of what's going on with the data, we have told the sankey function that the nodes it will be dealing with are in `graph.nodes` of our data structure.

I'm not sure what the `.layout(32);` portion of the code does, but I'd be interested to hear from any more knowledgeable readers. I've tried changing the values to no apparent affect and googling has drawn a blank. Internally to the `sankey.js` file it seems to indicate 'iterations' while it establishes computeNodeLinks, computeNodeValues, computeNodeBreadths, computeNodeDepths(iterations) and computeLinkDepths.

Then we add our links to the diagram with the following block of code;

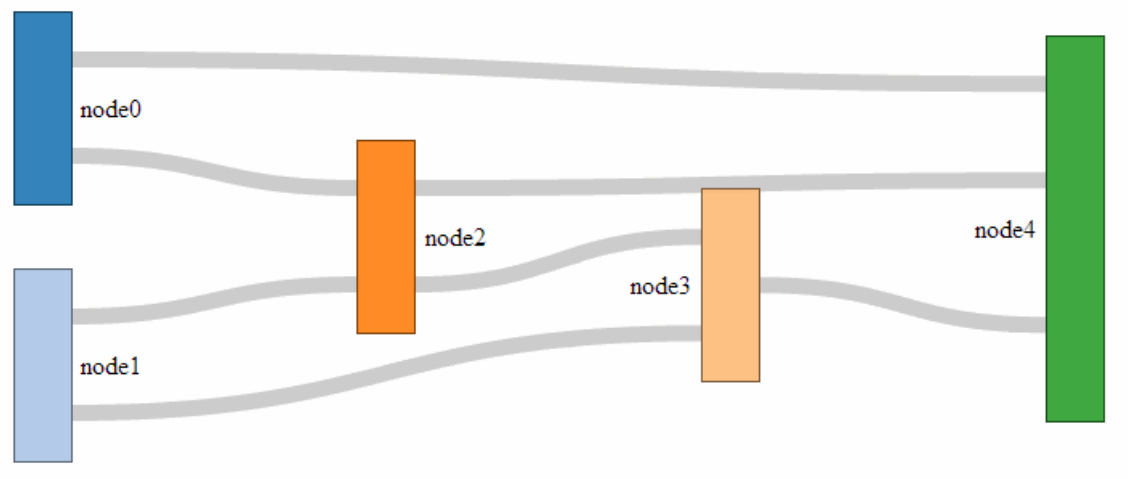
```
var link = svg.append("g").selectAll(".link")
    .data(graph.links)
  .enter().append("path")
    .attr("class", "link")
    .attr("d", path)
    .style("stroke-width", function(d) { return Math.max(1, d.dy); })
    .sort(function(a, b) { return b.dy - a.dy; });
```

This is an analogue of the block of code we examined way back in the section that we covered in explaining the code of our first simple graph.

We append svg elements for our links based on the data in `graph.links`, then add in the paths (using the appropriate CSS). We set the stroke width to the width of the value associated with each link or '1'. Whichever is the larger (by virtue of the Math.max function). As an interesting sideline, if we force this value to '10' thusly...

```
    .style("stroke-width", 10)
```

... the graph looks quite interesting.



stroke-width 10 for Sankey links

I have to admit that I don't know what the sort line (`.sort(function(a, b) { return b.dy - a.dy; });`) is supposed to achieve. Again, I'd be interested to hear from any more knowledgeable readers. I've tried changing the values to no apparent affect.

The next block adds the titles to the links;

```
link.append("title")
      .text(function(d) {
              return d.source.name + " □ " +
                      d.target.name + "\n" + format(d.value); });
```

This code appends a text element to each link when moused over that contains the source and target name (with a neat little arrow in between and the value) which, when applied with the `format` function adds the units.

The next block appends the node objects (but not the rectangles or text) and contains the instructions to allow them to be arranged with the mouse.

```
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));
```

While it starts off in familiar territory with appending the node objects using the `graph.nodes` data and putting them in the appropriate place with the `transform` attribute, I can only assume

that there is some trickery going on behind the scenes to make sure the mouse can do what it needs to do with the `d3.behaviour,drag` function. There is some excellent documentation on the wiki (https://github.com/mbostock/d3/wiki/Drag-behavior), but I can only presume that it knows what it's doing :-). The `dragmove` function is laid out at the end of the code, and we will explain how that operates later.

I really enjoyed the next block;

```
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); });
```

It starts off with a fairly standard appending of a rectangle with a height generated by its value ‘ { return d.dy; }and a `width dictated by the` sankey.jsfile `to fit the canvas` (.attr(“width”, sankey.nodeWidth())‘).

Then it gets interesting.

The colours are assigned in accordance with our earlier colour declaration and the individual colours are added to the nodes by finding the first part of the name for each node and assigning it a colour from the palate (the script looks for the first space in the name using a regular expression). For instance: ‘Widget X’, ‘Widget Y’ and ‘Widget’ will all be coloured the same even if the ‘Widget X’ and ‘Widget Y’ are inputs on the left and ‘Widget’ is a node in the middle.

The stroke around the outside of the rectangle is then drawn in the same shade, but `darker`. Then we return to the basics where we add the title of the node in a tool tip type effect along with the value for the node.

From here we add the titles 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");
```

Again, this looks pretty familiar. We position the text titles carefully to the left of the nodes. All except for those affected by the filter function (`return d.x < width / 2;`). Where if the position of the node on the x axis is less than half the width, the title is placed on the right of the node and anchored at the start of the text. Very neat.

The last block is also pretty neat, and contains a little surprise for those who are so inclined.

```
function dragmove(d) {
  d3.select(this).attr("transform",
     "translate(" + d.x + "," + (
              d.y = Math.max(0, Math.min(height - d.dy, d3.event.y))
          ) + ")");
  sankey.relayout();
  link.attr("d", path);
```

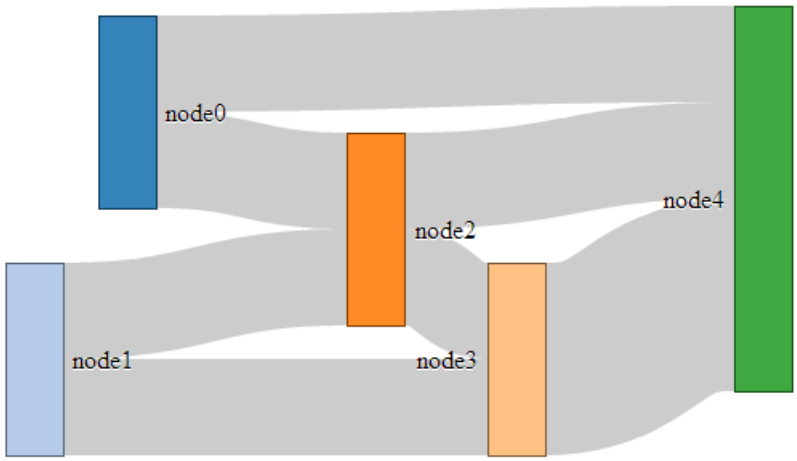
This declares the function that controls the movement of the nodes with the mouse. It selects the item that it's operating over (`d3.select(this)`) and then allows translation in the y axis while maintaining the link connection (`sankey.relayout(); link.attr("d", path);`).

But that's not the cool part. A quick look at the code should reveal that if you can move a node in the y axis, there should be no reason why you can't move it in the x axis as well!

Sure enough, if you replace the code above with this...

```
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);
```

... you can move your nodes anywhere on the canvas.



Move your nodes in x *and* y!

I know it doesn't seem to add anything to the diagram (in fact, it could be argued that there is a certain aspect of detraction) however, it doesn't mean that one day the idea doesn't come in handy :-). You can see a live version on github[68].

Formatting data for Sankey diagrams

From a JSON file with numeric link values

As explained in the previous section, data to form a Sankey diagram needs to be a combination of nodes and links.

```
{
"nodes":[
{"node":0,"name":"node0"},
{"node":1,"name":"node1"},
{"node":2,"name":"node2"},
{"node":3,"name":"node3"},
{"node":4,"name":"node4"}
],
"links":[
{"source":0,"target":2,"value":2},
{"source":1,"target":2,"value":2},
{"source":1,"target":3,"value":2},
{"source":0,"target":4,"value":2},
{"source":2,"target":3,"value":2},
{"source":2,"target":4,"value":2},
{"source":3,"target":4,"value":4}
]}
```

[68] http://bl.ocks.org/d3noob/5028304

As we also noted earlier, the "node" entries in the "nodes" section of the json file are superfluous and are really only there for our benefit since D3 will automatically index the nodes starting at zero. As a test to check this out we can change our data to the following;

```
{
"nodes":[
{"name":"Barry"},
{"name":"Frodo"},
{"name":"Elvis"},
{"name":"Sarah"},
{"name":"Alice"}
],
"links":[
{"source":0,"target":2,"value":2},
{"source":1,"target":2,"value":2},
{"source":1,"target":3,"value":2},
{"source":0,"target":4,"value":2},
{"source":2,"target":3,"value":2},
{"source":2,"target":4,"value":2},
{"source":3,"target":4,"value":4}
]}
```

(for reference this file is saved as sankey-formatted-names-and-numbers.json and the html file is Sankey-formatted-names-and-numbers.html)

This will produce the following graph;

Sankey graph with names

As you can see, essentially the same, but with easier to understand names.

As you can imagine, while the end result is great, the creation of the JSON file manually would be painful at best. Doing something similar but with a greater number of nodes / links would be a nightmare.

Let's see if we can make the process a bit easier and more flexible.

From a JSON file with links as names

It would make thing much easier, if you are building the data from hand, to have nodes with names, and the 'source' and 'target' links have those same name values as identifiers.

In other words a list of unique names for the nodes (and perhaps some details) and a list of the links between those nodes using the names for the nodes.

So, something like this;

```
{
"nodes":[
{"name":"Barry"},
{"name":"Frodo"},
{"name":"Elvis"},
{"name":"Sarah"},
{"name":"Alice"}
],
"links":[
{"source":"Barry","target":"Elvis","value":2},
{"source":"Frodo","target":"Elvis","value":2},
{"source":"Frodo","target":"Sarah","value":2},
{"source":"Barry","target":"Alice","value":2},
{"source":"Elvis","target":"Sarah","value":2},
{"source":"Elvis","target":"Alice","value":2},
{"source":"Sarah","target":"Alice","value":4}
]}
```

Once again, D3 to the rescue!

The little piece of code that can do this for us is here;

```
var nodeMap = {};
graph.nodes.forEach(function(x) { nodeMap[x.name] = x; });
graph.links = graph.links.map(function(x) {
  return {
    source: nodeMap[x.source],
    target: nodeMap[x.target],
    value: x.value
  };
});
```

This elegant solution comes from Stack Overflow[69] and was provided by Chris Pettitt (nice job).

So if we sneak this piece of code into here...

[69] http://stackoverflow.com/questions/14629853/json-representation-for-d3-networks

```
d3.json("data/sankey-formatted.json", function(error, graph) {

            //  <= Put the code here.

  sankey
      .nodes(graph.nodes)
      .links(graph.links)
      .layout(32);
```

... and this time we use our JSON file with just names (sankey-formatted-names.json) and our new html file (sankey-formatted-names.html) we find our Sankey diagram working perfectly!

Sankey graph with names again

Looking at our new piece of code...

```
var nodeMap = {};
graph.nodes.forEach(function(x) { nodeMap[x.name] = x; });
```

... the first thing it does is create an object called `nodeMap` (The difference between an array and an object in JavaScript is one that is still a little blurry to me and judging from online comments, I am not alone).

Then for each of the `graph.node` instances (where `x` is a range of numbers from `0` to the last node), we assign each node name to a number.

Then in the next piece of code...

```
graph.links = graph.links.map(function(x) {
  return {
    source: nodeMap[x.source],
    target: nodeMap[x.target],
    value: x.value
  };
```

... we go through all the links we have and for each link, we map the appropriate number to the correct name.

Very clever.

From a CSV with 'source', 'target' and 'value' info only.

In the first iteration of this section I had no solution to creating a Sankey diagram using a csv file as the source of the data.

But cometh the hour, cometh the man. Enter @timelyportfolio who, while claiming no expertise in D3 or JavaScript was able to demonstrate a solution[70] to exactly the problem I was facing! Well done Sir! I salute you and name the technique the timelyportfolio csv method!

So here's the cleverness that @timelyportfolio demonstrated;

Using a csv file (in this case called `sankey.csv`) that looks like this;

```
source,target,value
Barry,Elvis,2
Frodo,Elvis,2
Frodo,Sarah,2
Barry,Alice,2
Elvis,Sarah,2
Elvis,Alice,2
Sarah,Alice,4
```

We take this single line from our original Sankey diagram code;

```
d3.json("data/sankey-formatted.json", function(error, graph) {
```

And replace it with the following block;

[70] http://bl.ocks.org/timelyportfolio/5052095

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

  //set up graph in same style as original example but empty
  graph = {"nodes" : [], "links" : []};

    data.forEach(function (d) {
      graph.nodes.push({ "name": d.source });
      graph.nodes.push({ "name": d.target });
      graph.links.push({ "source": d.source,
                         "target": d.target,
                         "value": +d.value });
    });

    // return only the distinct / unique nodes
    graph.nodes = d3.keys(d3.nest()
      .key(function (d) { return d.name; })
      .map(graph.nodes));

    // loop through each link replacing the text with its index from node
    graph.links.forEach(function (d, i) {
      graph.links[i].source = graph.nodes.indexOf(graph.links[i].source);
      graph.links[i].target = graph.nodes.indexOf(graph.links[i].target);
    });

    //now loop through each nodes to make nodes an array of objects
    // rather than an array of strings
    graph.nodes.forEach(function (d, i) {
      graph.nodes[i] = { "name": d };
    });
```

The comments in the code (and they are fuller in @timelyportfolio's original gist solution[71]) explain the operation;

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

... Loads the csv file from the data directory.

```
  graph = {"nodes" : [], "links" : []};
```

... Declares `graph` to consist of two empty arrays called `nodes` and `links`.

[71] http://bl.ocks.org/timelyportfolio/5052095

```
    data.forEach(function (d) {
    graph.nodes.push({ "name": d.source });
    graph.nodes.push({ "name": d.target });
    graph.links.push({ "source": d.source,
                       "target": d.target,
                       "value": +d.value });
});
```

... Takes the `data` loaded with the csv file and for each row loads variables for the `source` and `target` into the `nodes` array then for each row loads variables for the `source` `target` and `value` into the `links` array.

```
graph.nodes = d3.keys(d3.nest()
  .key(function (d) { return d.name; })
  .map(graph.nodes));
```

... Is a routine that Mike Bostock described on Google Groups[72] that (as I understand it) nests each node name as a key so that it returns with only unique nodes.

```
graph.links.forEach(function (d, i) {
  graph.links[i].source = graph.nodes.indexOf(graph.links[i].source);
  graph.links[i].target = graph.nodes.indexOf(graph.links[i].target);
});
```

... Goes through each `link` entry and for each `source` and `target`, it finds the unique index number of that name in the nodes array and assigns the link source and target an appropriate number.

And finally...

```
graph.nodes.forEach(function (d, i) {
  graph.nodes[i] = { "name": d };
});
```

... Goes through each node and (in the words of @timelyportfolio) "*make nodes an array of objects rather than an array of strings*" (I don't really know what that means :-(. I just know it works :-).)

There you have it. A Sankey diagram from a csv file. Well played @timelyportfolio!

Both the html file for the diagram (`Sankey.formatted-csv.html`) and the data file (`sankey.csv`) can be found in the downloads section of d3noob.org.

[72] https://groups.google.com/forum/#!msg/d3-js/pl297cFtIQk/Eso4q_eBu1IJ

From MySQL as link information only automatically.

So, here we are. Faced with a dilemma of trying to get my csv formatted links into a Sankey diagram. In theory we need to go through our file, identify all the unique nodes and format the entire blob into JSON for use.

There must be a better way!

Well, I'm not going to claim that this is any better since it's a little like cracking a walnut with a sledgehammer. But to a man with just a sledgehammer, everything's a walnut.

So, let's use our newly developed MySQL and PHP skills to solve our problem. In fact, let's make it slightly harder for ourselves. Let's imaginge that we don't even *have* a value associated with our data, just a big line of source and target links. Something like this;

```
source,target
Barry,Elvis
Barry,Elvis
Frodo,Elvis
Frodo,Elvis
Frodo,Sarah
Frodo,Sarah
Barry,Alice
Barry,Alice
Elvis,Sarah
Elvis,Sarah
Elvis,Alice
Elvis,Alice
Sarah,Alice
Sarah,Alice
Sarah,Alice
Sarah,Alice
```

First thing first, just as we did in the example on using MySQL, import your csv file into a MySQL table which we'll call `sankey1` in database `homedb`.

Now we want to write a query that pulls out all the DISTINCT names that appear it the 'source' and 'target' columns. This will form our 'nodes' portion of the JSON data.

```
SELECT DISTINCT(`source`) AS name FROM `sankey1`
UNION
SELECT DISTINCT(`target`) AS name FROM `sankey1`
GROUP BY name
```

This query actually mashes two separate queries together where each returns DISTINCT instances of each `source` and `target` from the source and target columns. By default, the UNION operator eliminates duplicate rows from the result which means we have a list of each node in the table.

name
Barry
Frodo
Elvis
Sarah
Alice

Sankey nodes from MySQL

Exxxeellennt……. (channelling Mr Burns)

Now we run a separate query that pulls out each distinct 'source' and 'target' combination and the number of times (COUNT(*)) that it occurs.

```
SELECT `source` AS source, `target`  as target, COUNT(*) as value
FROM `sankey1`
GROUP BY source, target
```

This query gets all the sources plus all the targets and groups them by first the source and then the target. Each line is therefore unique and the `COUNT(*)` sums up the number of times that each unique combination occurs.

source	target	value
Barry	Alice	2
Barry	Elvis	2
Elvis	Alice	2
Elvis	Sarah	2
Frodo	Elvis	2
Frodo	Sarah	2
Sarah	Alice	4

Sankey links from MySQL

That was surprisingly easy wasn't it?

MySQL is good for simple jobs, but we are of course a long way from finished since at this stage all we have is what looks like two tables in a spreadsheet.

So now we turn to PHP.

Remember from our previous exposure, we described PHP as the glue that could connect parts of web pages together. In this case we will use it to glue our MySQL database to our JavaScript.

We need to carry out our queries and return the information in a format that d3.js can understand. In this instance we will select JSON as it's probably the most ubiquitous, and it suits the format of our original manual data.

Let's cut to the chase and look at the code:

```
<?php
    $username = "homedbuser";
    $password = "homedbuser";
    $host = "localhost";
    $database="homedb";

    $server = mysql_connect($host, $username, $password);
    $connection = mysql_select_db($database, $server);

    $myquery = "
SELECT DISTINCT(`source`) AS name FROM `sankey1`
UNION
SELECT DISTINCT(`target`) AS name FROM `sankey1`
GROUP BY name
";
    $query = mysql_query($myquery);

    if ( ! $myquery ) {
        echo mysql_error();
        die;
    }

    $nodes = array();

    for ($x = 0; $x < mysql_num_rows($query); $x++) {
        $nodes[] = mysql_fetch_assoc($query);
    }

    $myquery = "
SELECT `source` AS source, `target`  as target, COUNT(*) as value
FROM `sankey1`
GROUP BY source, target
";
    $query = mysql_query($myquery);

    if ( ! $myquery ) {
        echo mysql_error();
       die;
    }

    $links = array();

    for ($x = 0; $x < mysql_num_rows($query); $x++) {
        $links[] = mysql_fetch_assoc($query);
    }
```

```
echo "{";
echo '"links": ', json_encode($links), "\n";
echo ',"nodes": ', json_encode($nodes), "\n";
echo "}";

    mysql_close($server);
?>
```

Astute readers will recognise that this is very similar to the script that we used to extract data from the MySQL database for generating a simple line graph. If you haven't checked it out, and you're unfamiliar with PHP, you will want to read that section first.

We declare all the appropriate variables which we will use to connect to the database. We then connect to the database and run our query.

After that we store the nodes data in an array called `$nodes`.

Then we run our second query (we don't close the connection to the database since we're not finished with it yet).

The second query returns the link results into a second array called `$links` (pretty imaginative).

Now we come to a part that's a bit different. We still need to echo out the data in the same way we did in our line graph, but in this case we need to add the data together with the associated `links` and `nodes` identifiers.

```
echo "{";
echo '"links": ', json_encode($links), "\n";
echo ',"nodes": ', json_encode($nodes), "\n";
echo "}";
```

(if you look closely, the syntax will produce our JSON formatted output).

At last, we need to call this PHP script from our html file in the same way that we did for the line graph. So amend the html file to change the loading of the JSON data to be from our PHP file thusly;

```
d3.json("php/sankey.php", function(error, graph) {
```

And there you have it! So many ways to get the data.

Both the PHP file (sankey.php) and the html file (sankey-mysql-import.html) are available in the downloads section on d3noob.org.

Sankey diagram case study

Armed with all this new found knowledge on building Sankey diagrams, what can you do?

Well, I suppose it all depends on your data set, but remember, Sankey diagrams are good at flows, but they won't do loops / cycles easily (although there has been some good work done in this direction here http://bl.ocks.org/cfergus/3956043[73] and here http://bl.ocks.org/kunalb/4658510[74]).

So let's choose a flow.

In this case we'll selected the flow of data that represents a view of global, anthropogenic greenhouse gas (GHG) emissions. The image is an alternative to the excellent diagram on the World Resources Institute (http://www.wri.org/chart/world-greenhouse-gas-emissions-2005) and as such my version pales in comparison to theirs.

However, the aim is to play with the technique, not to emulate :-).

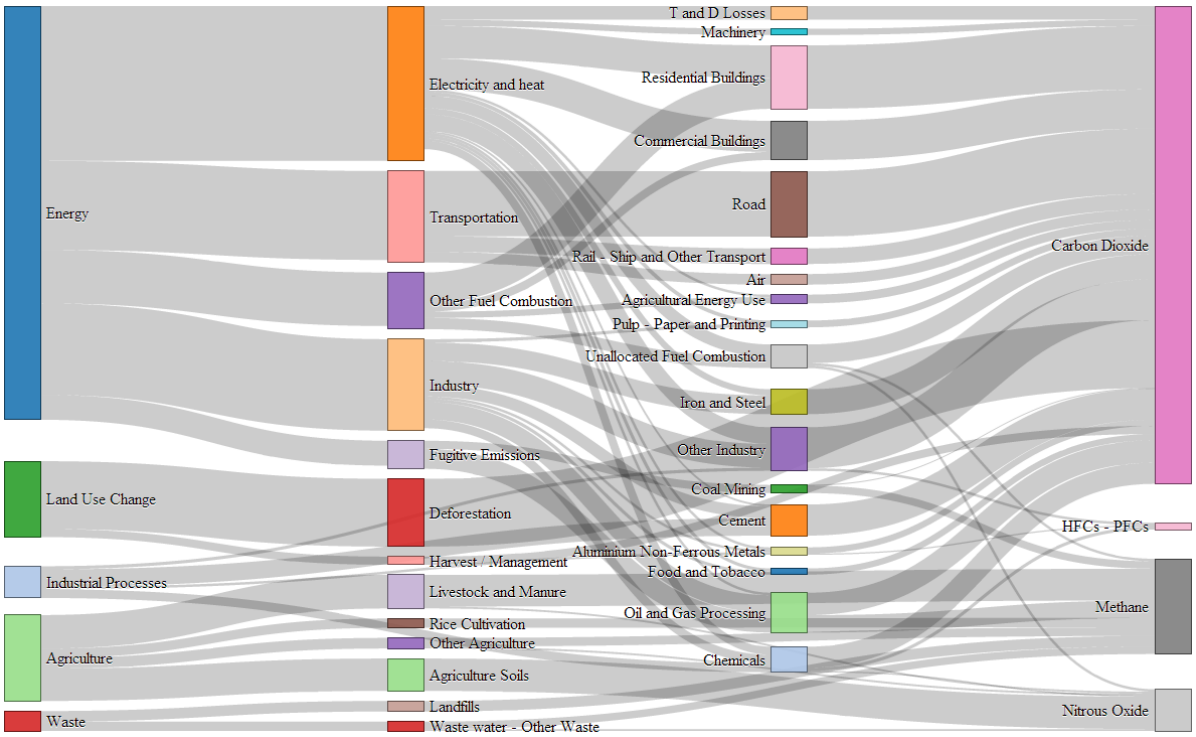
So starting with the data presented in the original diagram, we have to capture the links into a csv file. I did this the hard way (since there didn't appear to be an electronic version of the data) by reading the graph and entering the figures into a csv file. From here we import it into our MySQL database and then convert it into sankey formatted JSON by using our PHP script that we played with in the example of extracting information from a MySQL database. In this case instead of needing to perform a `COUNT(*)` on the data, it's slightly easier since the value is already present.

Because we want this diagram to be hosted on Gist and accessible on bl.ocks.org, we run the PHP file directly into the browser so that it just shows the JSON data on the screen. We save this file with the suffix `.json` and we have our data (in this case the file is named `sankeygreenhouse.json`).

We amend our html file to look at our new `.json` file and voila!

[73] http://bl.ocks.org/cfergus/3956043

[74] http://bl.ocks.org/kunalb/4658510



Sankey diagram of greenhouse gas emissions in 2005

Sankeytastic!

You can find this as a live example and with all the code and data on bl.ocks.org[75].

[75] http://bl.ocks.org/d3noob/5015397

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 it 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[76] and the slides here: http://mbostock.github.com/d3/talk/20110921/#0[77]. 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

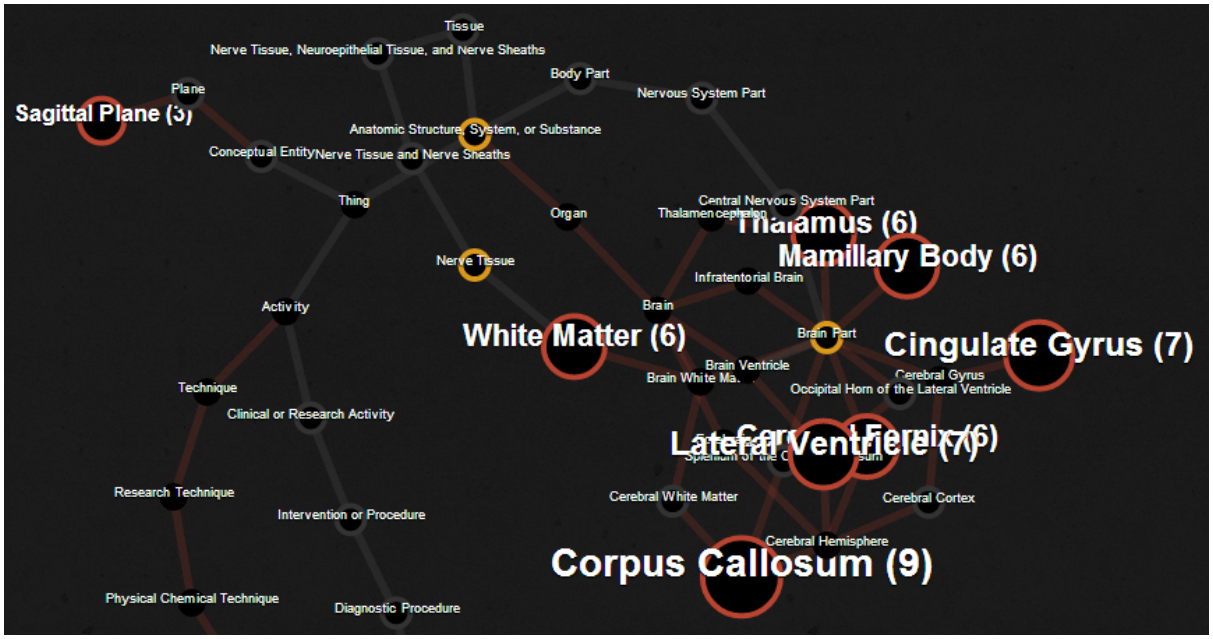
Simultaneous forces of repulsion and multiple gravitational focus points creates a natural clustering of data points (Source: Mike Bostock http://bl.ocks.org/mbostock/1249681[78]). The graph is animated, so the artefacts such as overlapping circles and the purple circle that is located beside the red area are transitory.

Force Directed Graph with Pan / Zoom

[76] http://vimeo.com/29458354

[77] http://mbostock.github.com/d3/talk/20110921/#0

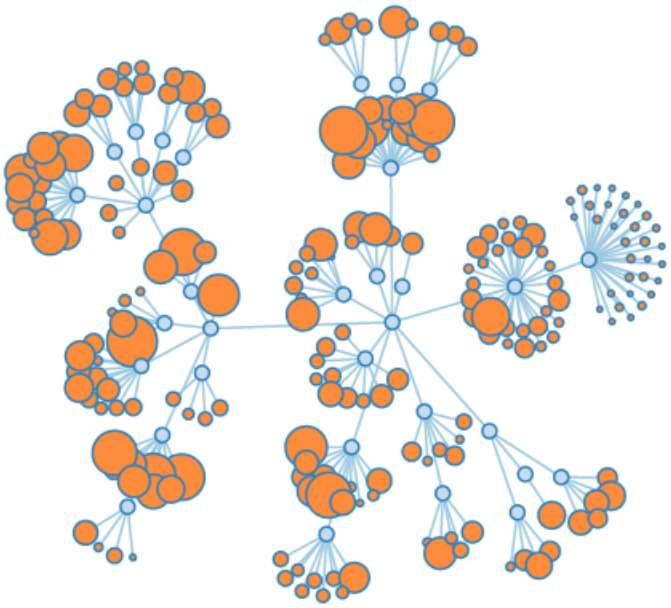
[78] http://bl.ocks.org/mbostock/1249681



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/[79].

Collapsible Force Layout



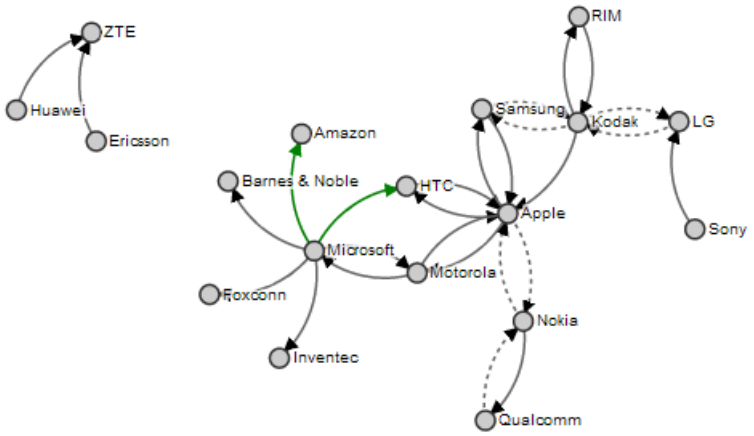
Collapsible Force Layout

This force directed graph can have individual nodes expanded or collapsed by clicking on them

[79] http://graus.nu/blog/force-directed-graphs-playing-around-with-d3-js/

to reveal or hide greater detail (Source: Mike Bostock http://bl.ocks.org/mbostock/1062288[80]).

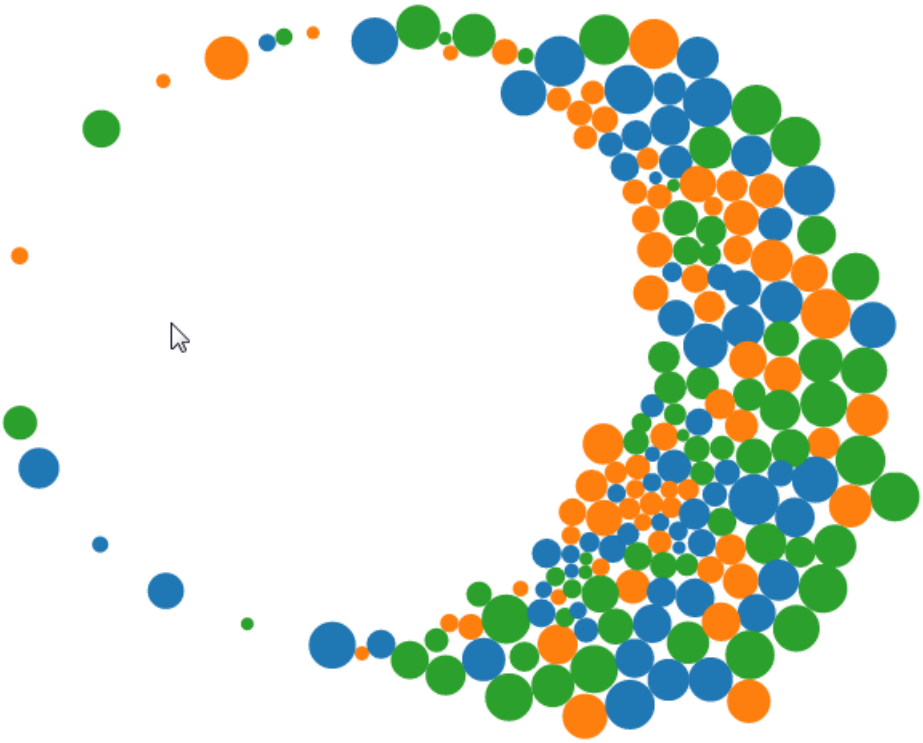
Force Directed Graph showing Directionality



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[81]).

Collision Detection

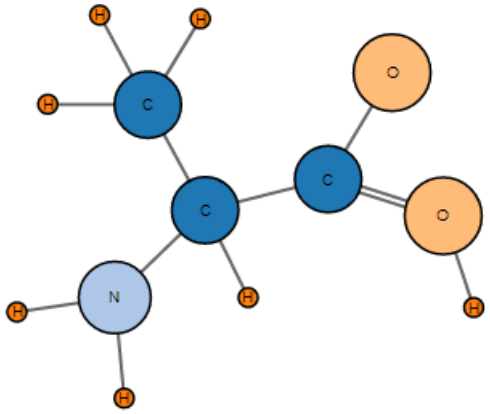


Collision Detection

[80] http://bl.ocks.org/mbostock/1062288
[81] http://bl.ocks.org/mbostock/1153292

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[82]).

Molecule Diagram



Molecule Diagram

Just for fun, here is a diagram the 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[83]).

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[84].

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 it's 'great attractor' and avoids what would otherwise be a small black hole type effect.

Friction

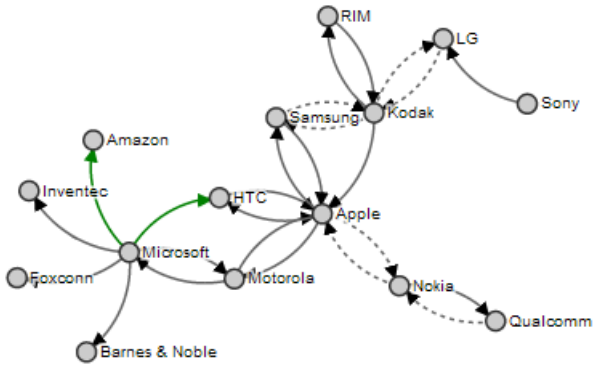
The frictional force is one designed to act on the movement of a node to reduce it's speed over time. It isn't implemented as true friction (in the physical sense) and should be thought of as a 'velocity decay' in the truer sense.

[82] http://bl.ocks.org/mbostock/3231298
[83] http://bl.ocks.org/mbostock/3037015
[84] https://github.com/mbostock/d3/wiki/Force-Layout

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.

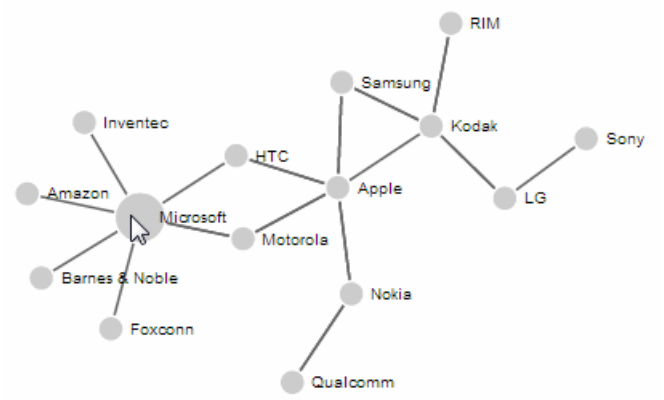
Force directed graph examples.

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 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[85] graph....



Mobile Patent Suits

... for the directionality and link encoding and the Force-Directed Graph with Mouseover[86] graph...

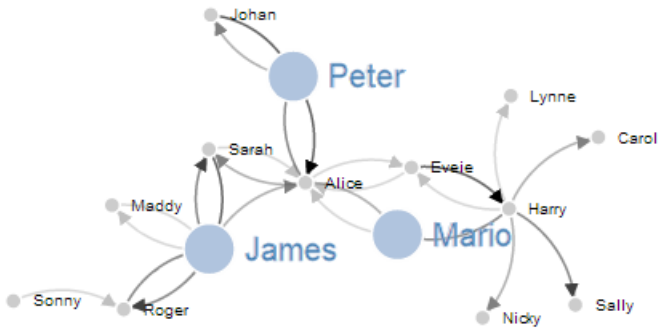


Force-Directed Graph with Mouseover

... for the mouseover effects (note the enlarged 'Microsoft' circle).

[85] http://bl.ocks.org/mbostock/1153292
[86] http://bl.ocks.org/mbostock/2706022

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;



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 this graph can be found on bl.ocks.org[87].

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 peoples names and associated values (to represent the strength of the links between the two). 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
Maddy,James,0.3
Sonny,Roger,0.5
```

[87] http://bl.ocks.org/d3noob/5155181

```
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 type="text/javascript" src="d3/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("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();

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 type="text/javascript" src="d3/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`) from our `data` directory.

```
d3.csv("data/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 out 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[88], 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[89]). Put simply, it sets the nodes to the `nodes` we have previously set in our object.

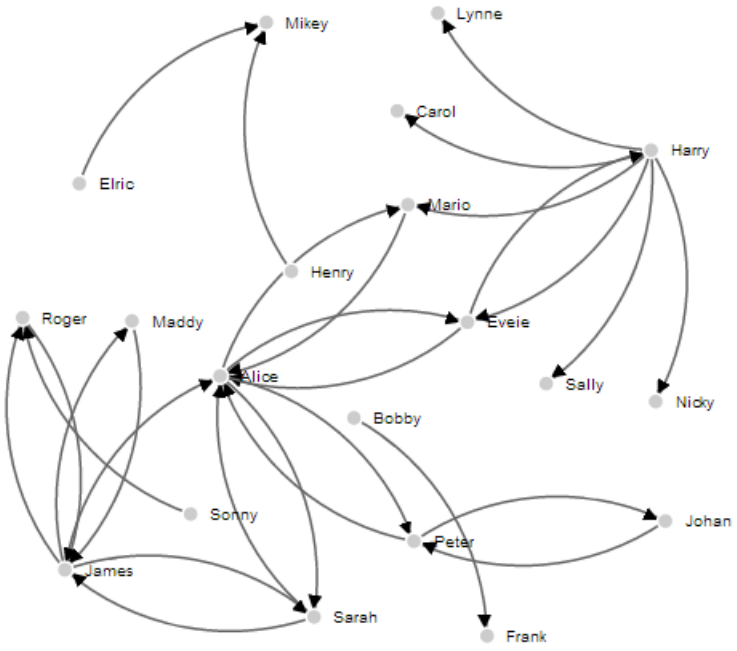
`.links(links)` does for links what `.nodes` did for nodes.

`.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.

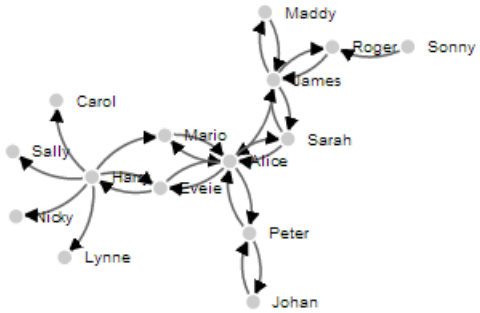
Setting this value (and other force values) can be something of a balancing act. For instance, here is the result of setting the `.linkDistance` to 160.

[88] https://github.com/mbostock/d3/wiki/Force-Layout
[89] https://github.com/mbostock/d3/wiki/Arrays#wiki-d3_values



Link distance set to 160

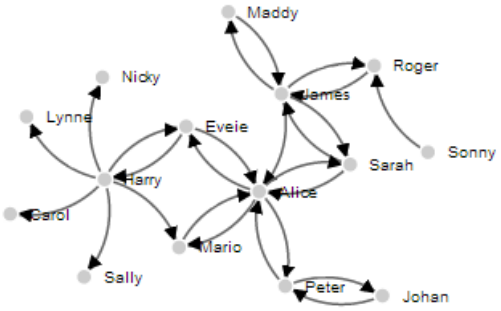
Here the charged nodes are trying to arrange themselves at an appropriate distance, but the length of the links means that their arrangement is not very pretty. Likewise if we change the value to 30 we get the following;



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 these progression of steps that give 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[90] and http://www.w3schools.com/svg/svg_reference.asp?[91]. 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 the encode the value as 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.

[90] http://www.w3.org/TR/SVG/coords.html#ViewBoxAttribute
[91] 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.

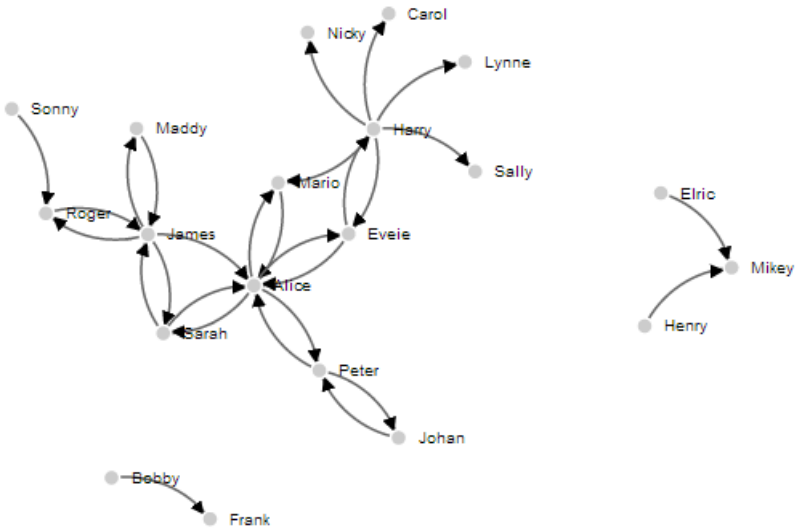
```
node.append("text")
    .attr("x", 12)
    .attr("dy", ".35em")
    .text(function(d) { return d.name; });
```

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

The code and data for this example can be found as 'Basic Directional Force Layout Diagram[92]' on bl.ocks.org.

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;

[92] http://bl.ocks.org/d3noob/5141278

```
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 (`d3.select(this).select("text").transition()`).

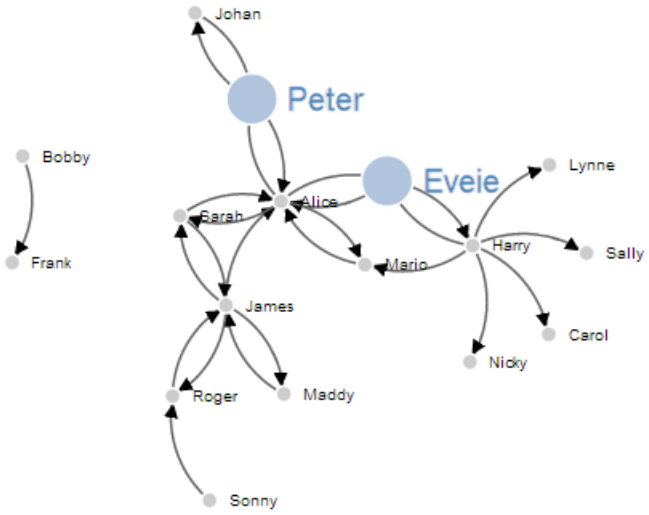
Then we define the new properties that will be in place after the transition. We move the text's x position (`.attr("x", 22)`), make the text fill steel blue (`.style("fill", "steelblue")`), set the stroke around the edge of the text light steel blue (`.style("stroke", "lightsteelblue")`), set that stroke 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 code and data for this example can be found as Directional Force Layout Diagram with Node Highlighting[93] on bl.ocks.org.

Directional Force Layout Diagram (varying link opacity)

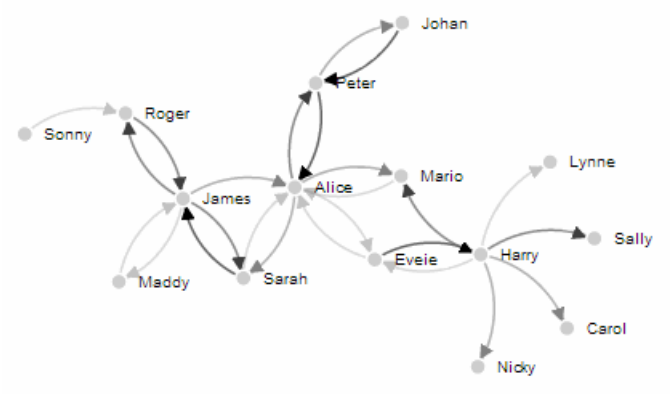
The next variation to our force layout diagram is the addition of variation in the link to represent different values (think of the number of packets passed or the amount of money transferred).

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 0.25, 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;

[93] http://bl.ocks.org/d3noob/5141528



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")
```

...to 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 code and data for this example can be found as Directional Force Layout Diagram with varying link opacity[94] on bl.ocks.org.

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.

[94] http://bl.ocks.org/d3noob/5155181

Mapping with d3.js

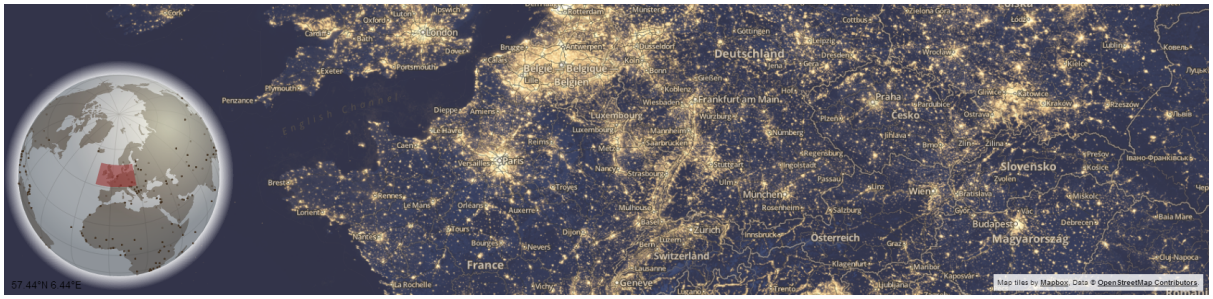
Another string to the bow of d3.js is the addition of a set of powerful routines for handling geographical information.

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[95] provides a glimpse into Jason's expertise and no visit would be complete without marvelling at his work with geographic projections[96].

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[97] 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

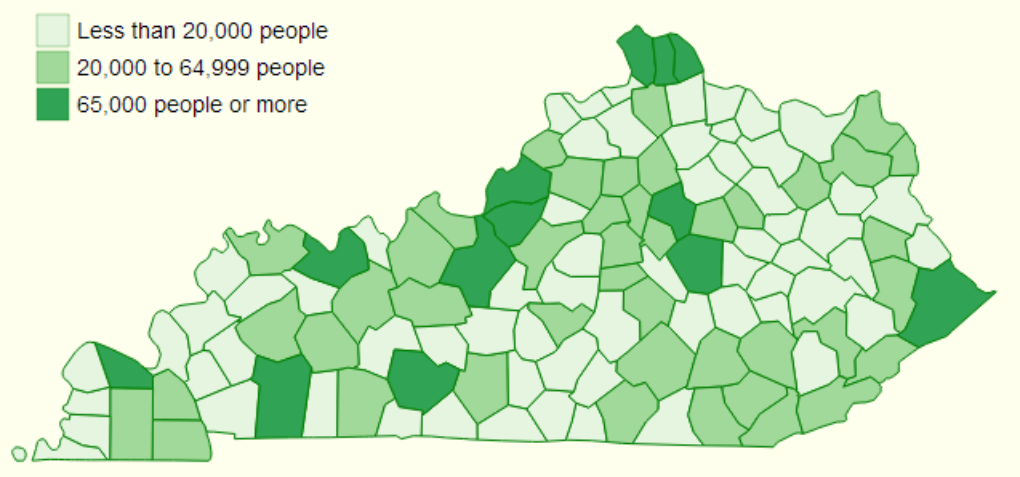
Above is an interactive visualization showing the position of the main map on a faux D3 3d globe with a Mapbox / Open Street Map main window. Source dev.geosprocket.com[98] Source Bill Morris.

[95] http://www.jasondavies.com/

[96] http://www.jasondavies.com/maps/

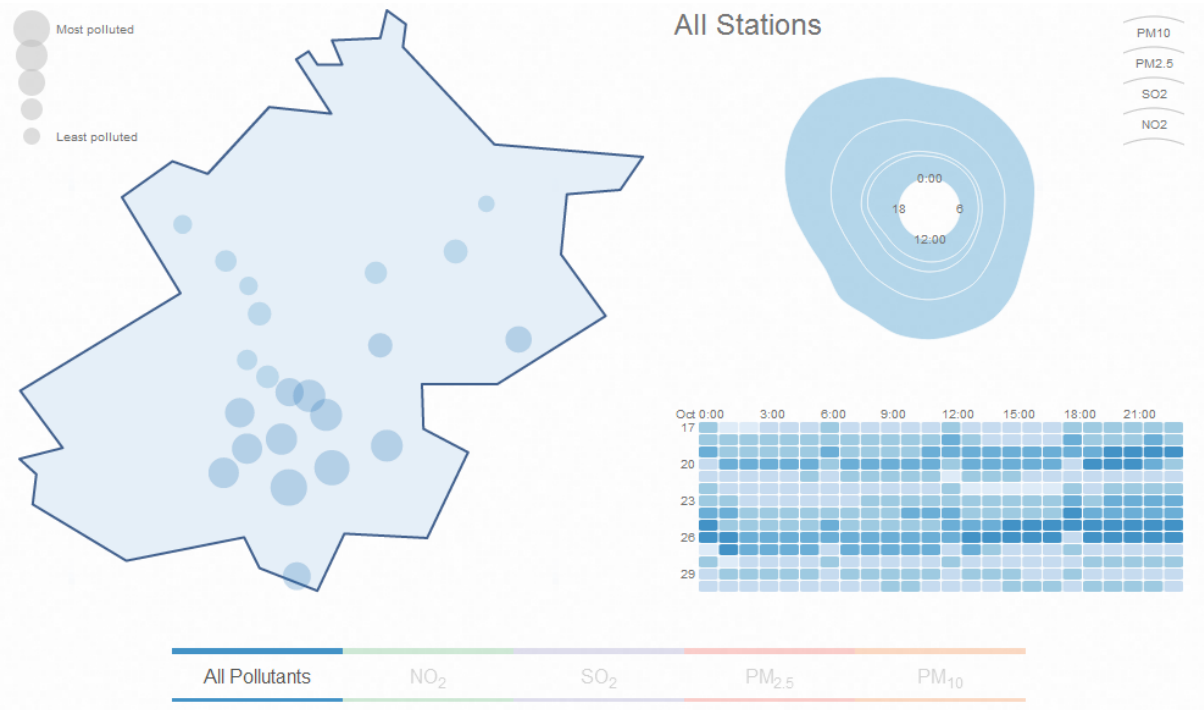
[97] http://biovisualize.github.com/d3visualization/#visualizationType=map

[98] http://dev.geosprocket.com/d3/finder/



Kentucky Count Population from the 2010 census

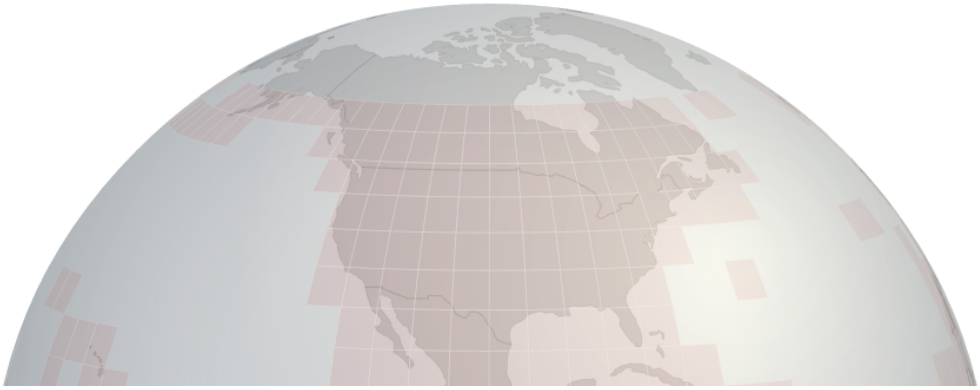
This is a breakdown of population in Kentucky Counties from the 2010 census. Source: ccarpenterg.github.com[99] by Cristian Carpenter.



Beijing air pollution

This map visualizes air pollution in Beijing. Source: scottcheng.github.com[100] by Scott Cheng.

[99] http://ccarpenterg.github.com/blog/us-census-visualization-with-d3js/
[100] http://scottcheng.github.com/bj-air-vis/



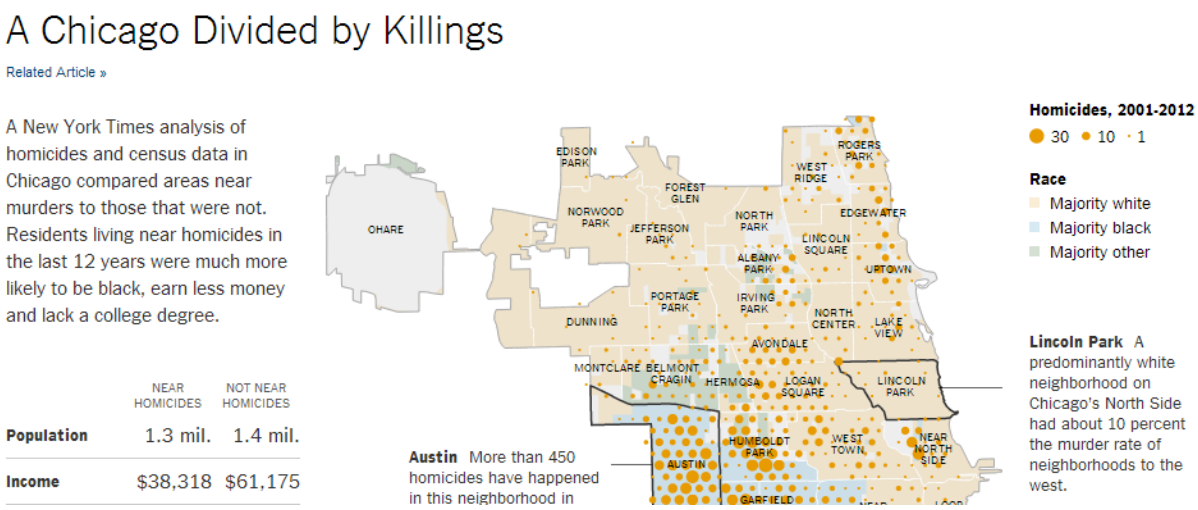
Shuttle Radar Topography Mission tile downloading

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[101] by Derek Watkins.

Animated World tour

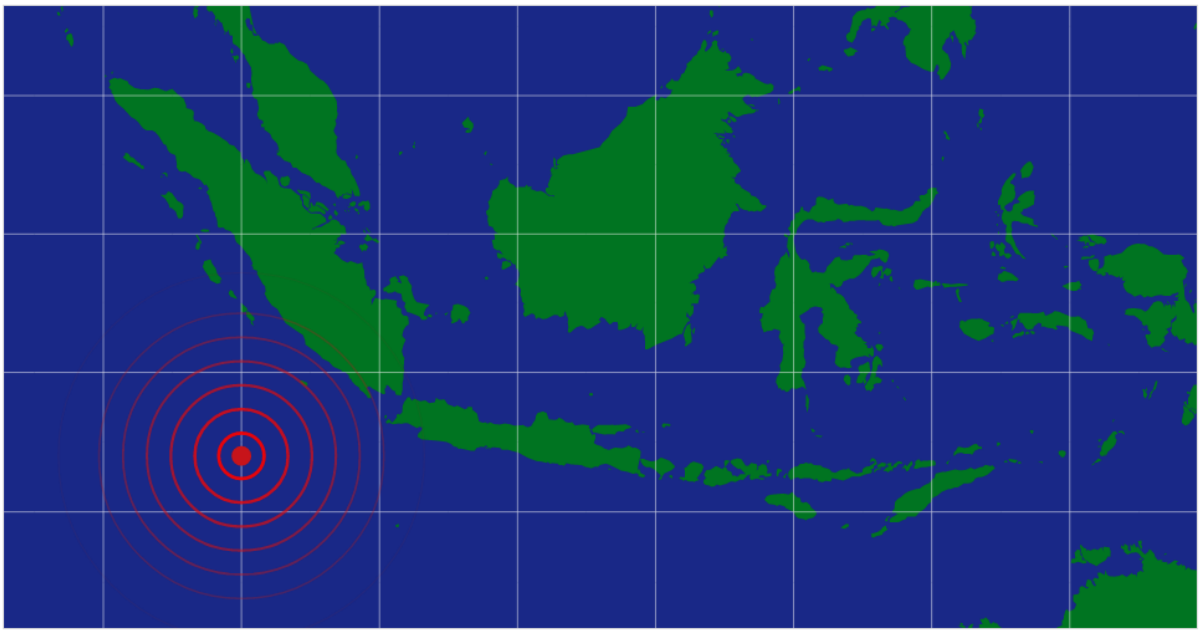
This is a static screen-shot of an animated tour of the Worlds countries. Source bl.ocks.org[102] by Mike Bostock.

[101] http://dwtkns.com/srtm/
[102] http://bl.ocks.org/mbostock/4183330



A Chicago Divided by Killings: New Your Times

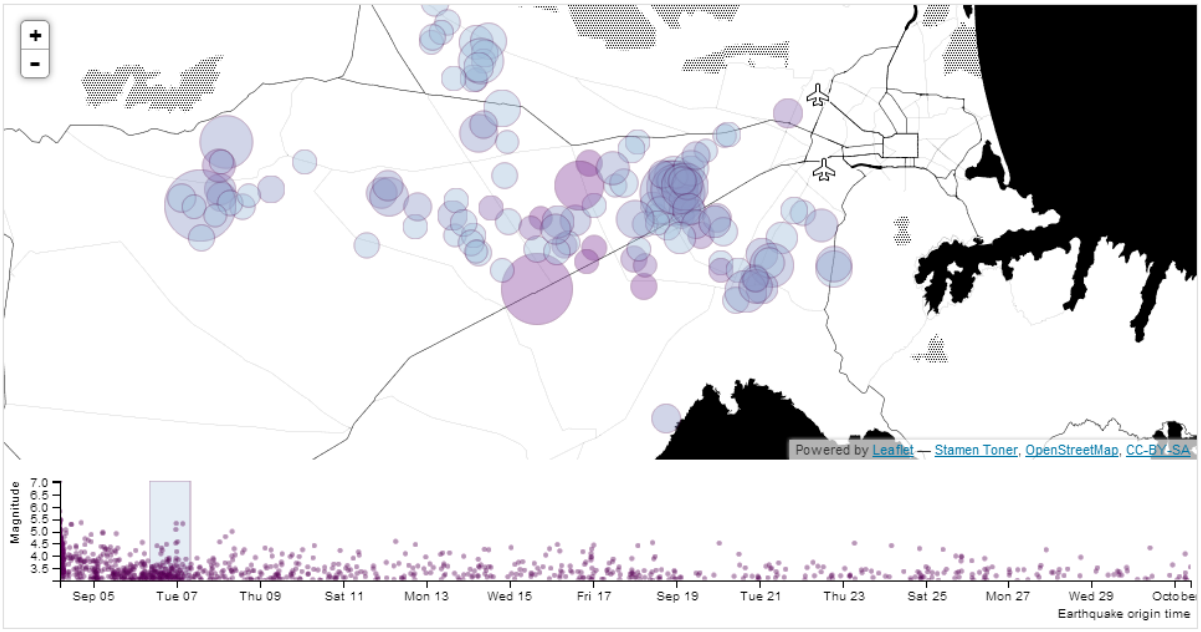
This is one of the great infographics published by the New York Times[103]. Source: www.nytimes.com[104] by Mike Bostock, Shan Carter and Kevin Quealy.



Concentric circles emanating from glowing red dot

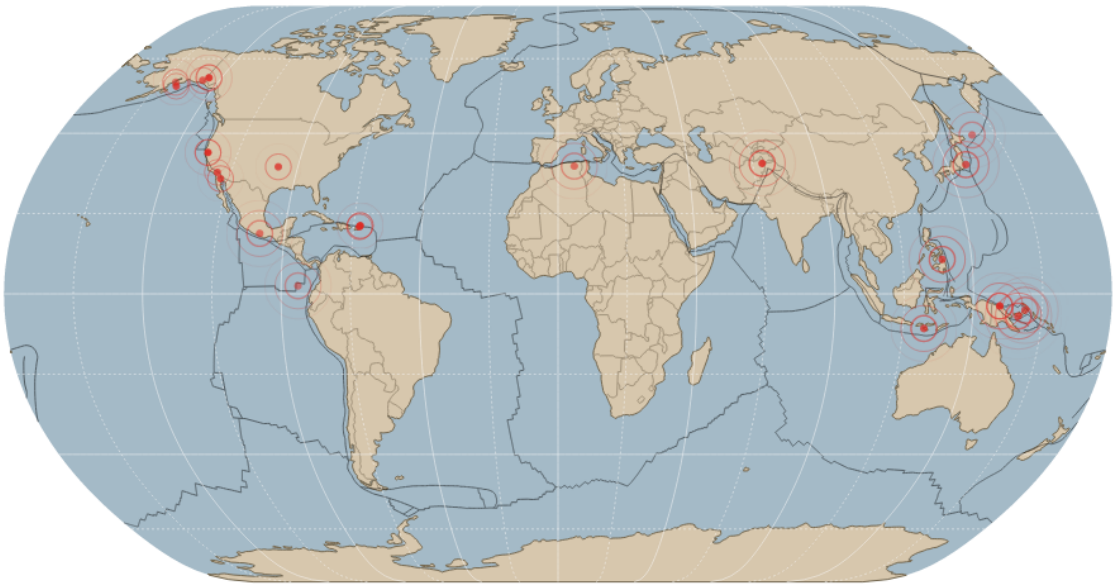
This is an animated graphic showing a series of concentric circles emanating from glowing red dot which was styled after a news article in The Onion[105]. Source: bl.ocks.org[106] by Mike Bostock.

[103] http://www.nytimes.com
[104] http://www.nytimes.com/interactive/2013/01/02/us/chicago-killings.html?_r=0
[105] http://www.theonion.com/video/breaking-news-series-of-concentric-circles-emanati,14204/
[106] http://bl.ocks.org/mbostock/4503672



Christchurch earthquakes timeline

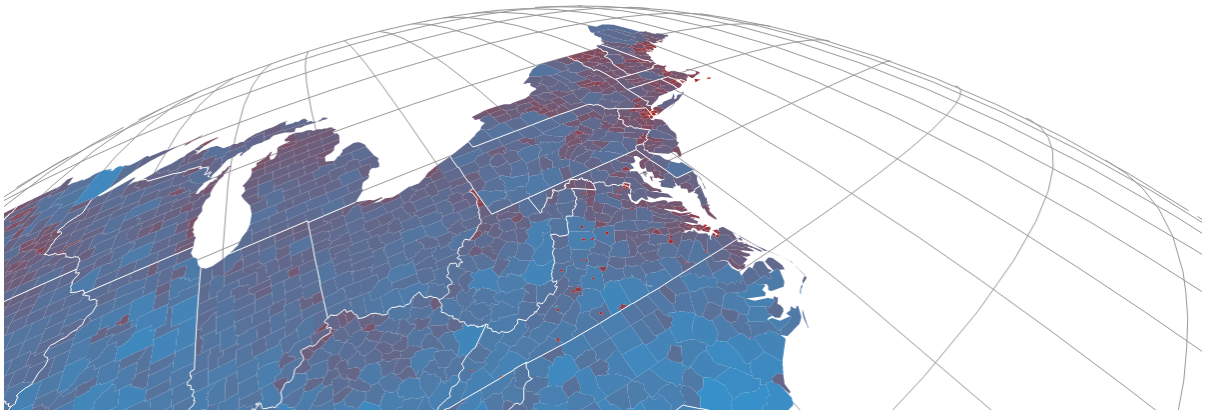
Here we see earthquakes represented on a selectable timeline where D3 generates a svg overlay and the map layer is created using Leaflet. Source: bl.ocks.org[107] by tnightingale.



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[108] by benelsen.

[107] http://bl.ocks.org/tnightingale/4718717
[108] http://bl.ocks.org/benelsen/4969007



Satellite projection

An interactive satellite projection. Source dev.geosprocket.com[109] 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 it's 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[110].

[109] http://dev.geosprocket.com/d3/sat/

[110] 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).

Here is the full code;

```
<!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)
});


</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 scree 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 type="text/javascript" src="d3/d3.v3.js"></script>
<script src="js/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.

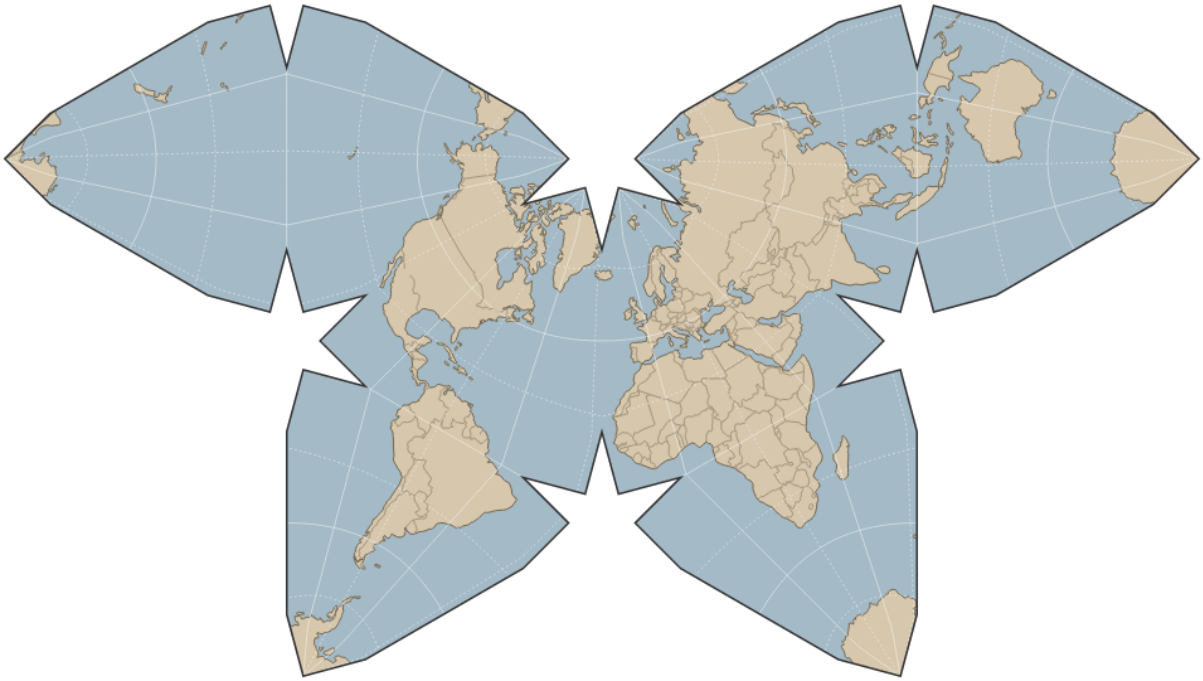
```
var width = 960,
    height = 500;
```

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(900)
    .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[111].



The Waterman Butterfly

There are a lot of alternatives available. Please have a browse on the wiki[112] 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.

[111] http://bl.ocks.org/mbostock/4458497

[112] https://github.com/mbostock/d3/wiki/Geo-Projections

center

If center is specified, this sets the projection's center to the specified location as 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 (`0`) for longitude (left to right) and `5` degrees North of the equator (North is positive values of latitude, South is negative values). 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, 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 follow 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 spcify 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("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

The code and data for this example can be found as World Map Centered on the Pacific[113] on bl.ocks.org.

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[114].

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.

[113] http://bl.ocks.org/d3noob/5189184

[114] https://github.com/mbostock/d3/wiki/Zoom-Behavior

```
svg.call(zoom)
```

Then we relax and explore our map!

The World map with zoom and pan

The code and data for this example can be found as World Map with zoom and pan[115] on bl.ocks.org.

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[116] 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;

[115] http://bl.ocks.org/d3noob/5189284
[116] http://www.openstreetmap.org/

```
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
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("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("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");
```

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("data/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("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 code and data for this example can be found as World map with zoom / pan and cities[117] on bl.ocks.org.

Additionally the full code can be found in the appendix section at the rear of the book.

[117] http://bl.ocks.org/d3noob/5193723

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[118] 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 it's 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 :-).

[118] http://bl.ocks.org/

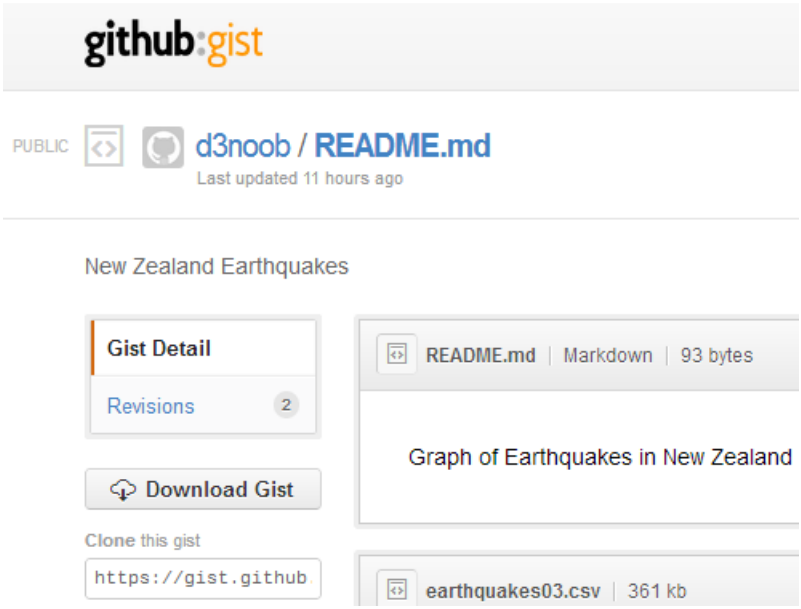
Installing the plug-in for bl.ocks.org for easy block viewing

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.

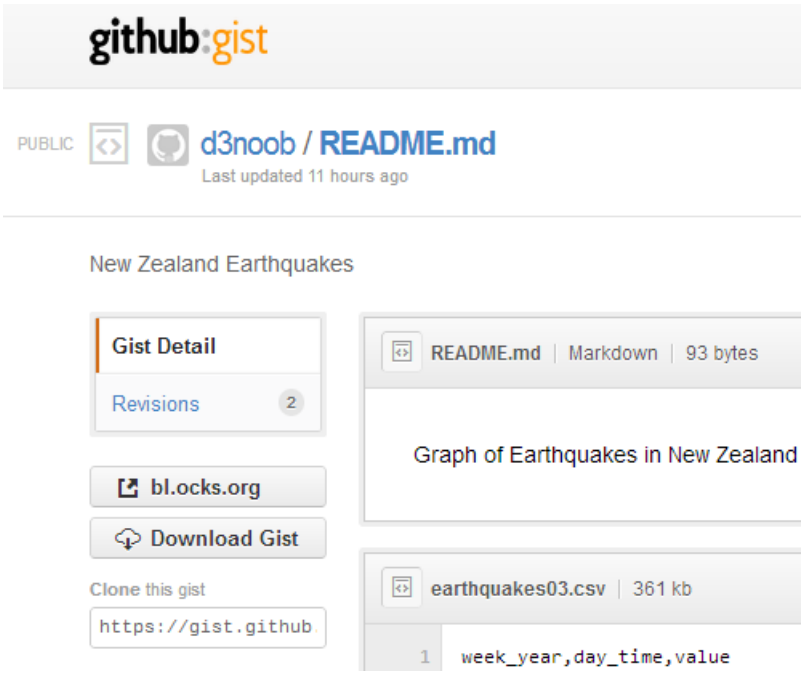
Well, throw away that tiresome technique! It's time to move into the 21st century with some plug-in goodness. Clever Mike Bostock has put together some handy dandy browser extensions that will add a button to your Chrome, Safari or Firefox browser to take you straight from your Gist to your block!

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[119] or from theGithub page[120] 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[121].

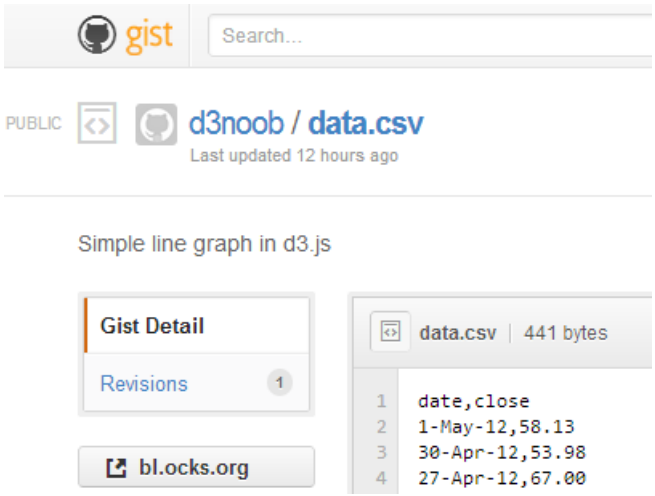
Setting the scene:

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.

[119] http://bl.ocks.org/

[120] https://github.com/mbostock/bl.ocks.org

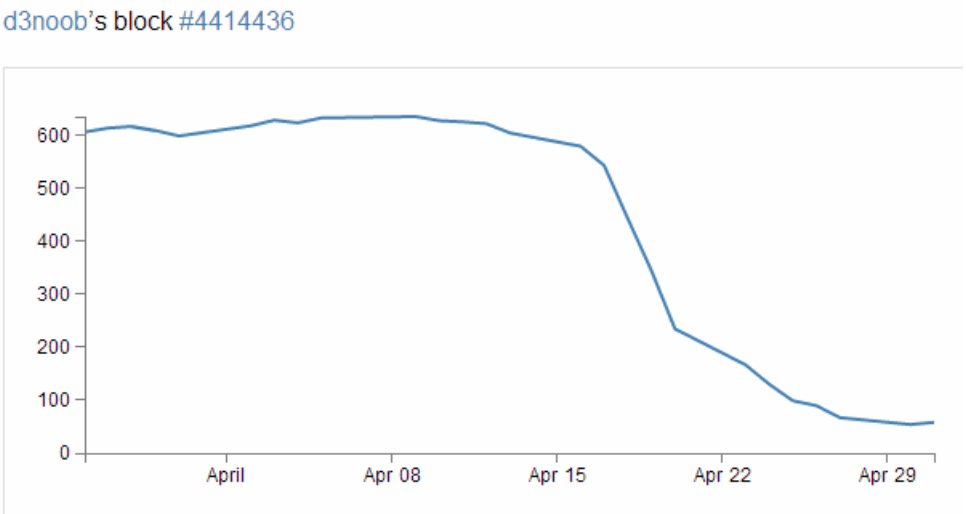
[121] https://groups.google.com/forum/?fromgroups=#!topic/d3-js/FBosXiTB9Pc



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.

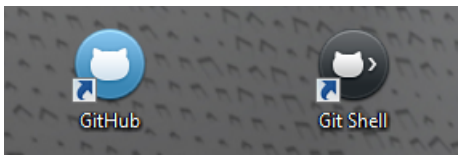
Hang on, what had Mike said on the bl.ocks.org main page?

"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

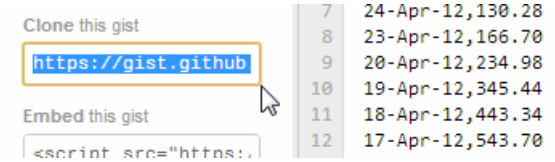
Enough of the scene setting. Let's git going :-).

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[122].

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.

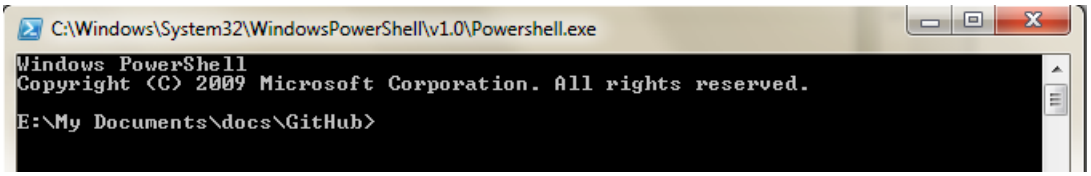
[122] https://groups.google.com/forum/?fromgroups=#!topic/d3-js/FBosXiTB9Pc



Copy the 'Clone this gist' link

(this should look something like https://gist.github.com/441443[123])

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;

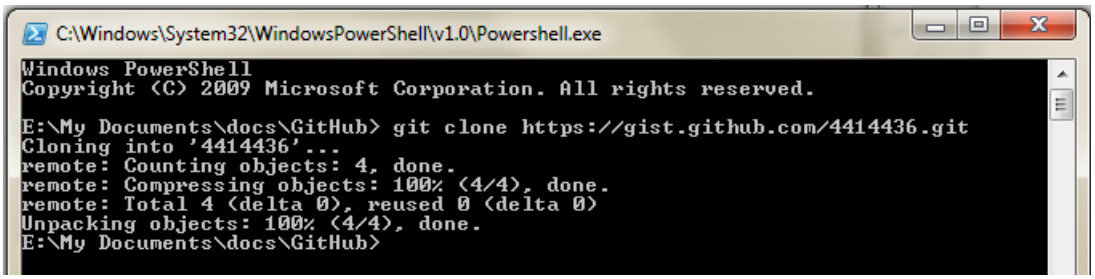


The Git Shell is open for business

You can clone the gist to a local folder with the command;

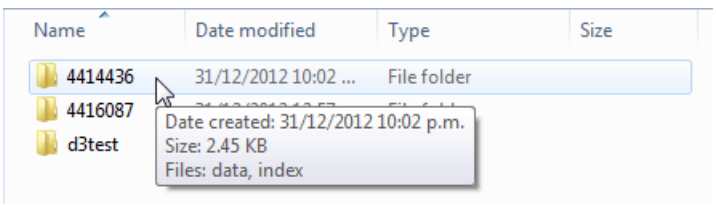
```
git clone https://gist.github.com/4414436.git
```

(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

[123] https://gist.github.com/441443

And there it is (Ooo… Look almost New Years!).

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"
```

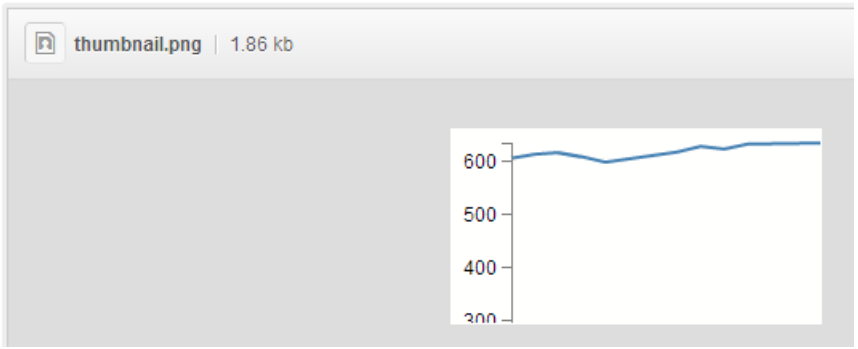
Running the git commit command

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;

```
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/[124]
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` It will add a folder with the gist id as a name (i.e., 4414436) under the current working directory.

[124] 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 test.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.

Appendix: Simple Line Graph

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


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>
<script type="text/javascript" src="d3/d3.v3.js"></script>
<script>


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.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; })]);
    svg.append("path")               // Add the valueline path.
        .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);
});


</script>
</body>
```

Appendix: 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;
}
line {
    stroke: grey;
}
.axis path,
.axis line {
    fill: none;
    stroke: grey;
    stroke-width: 1;
    shape-rendering: crispEdges;
}
.grid .tick {
    stroke: lightgrey;
    opacity: 0.7;
}
.grid path {
          stroke-width: 0;
}
.area {
    fill: lightsteelblue;
      stroke-width: 0;
}
```

```
</style>
<body>
<script type="text/javascript" src="d3/d3.v3.js"></script>
<script>


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 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.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; })]);
    // 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("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);
    // Add a the text label white background 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>
```

Appendix: 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>
```

Appendix: 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);
?>
```

Appendix: 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>
```

Appendix: 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>
```

Appendix: 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>
```