Chapter No. 1
"Getting Started with d3.js"
In this package, you will find:

A Biography of the author of the book
A preview chapter from the book, Chapter NO.1 "Getting Started with d3.js"
A synopsis of the book’s content
Information on where to buy this book

About the Author

Swizec Teller is a geek with a hat. Founding his first startup at 21, he is now looking for the next big idea as a full-stack web generalist focusing on freelancing for early-stage startup companies.

When he isn't coding, he's usually blogging, writing books, or giving talks at various non-conference events in Slovenia and nearby countries. He is still looking for a chance to speak at a big international conference.

In November 2012, he started writing Why Programmers Work at Night, and set out on a quest to improve the lives of developers everywhere.

I want to thank @gandalfar and @robertbasic for egging me on while writing and being my guinea pigs for the examples. I also want to send love to everyone at @psywerx for keeping me sane and creating one of the best datasets ever.

For More Information:

Data Visualization with d3.js:

When learning d3.js on your own, there is often a feeling of Step 1: Draw two circles, Step 2: Draw the rest of the owl. This book tries to bridge that gap.

It uses complete examples that take you from basic shapes on a page to full-blown examples. There is no magic here, no steps are left unexplained. You will understand everything that goes into making a visualization with d3.js.

We'll touch everything from manipulating data to make it easier to work with, to using advanced features to separate drawing from calculating coordinates.

What This Book Covers

Chapter 1, Getting Started with d3.js, gives a simple example to show you the basics of d3.js and helps you to set up a common environment, which is used throughout the rest of the book.

Chapter 2, A Primer on DOM, SVG, and CSS, explains in detail how to use d3.js for manipulating content on a page, paying special attention to SVG and the core tools for creating images.

Chapter 3, Making Data Useful, shows you how to manipulate data in a functional manner, load data from external sources, and use the built-in tools of d3.js to avoid tedious coding.

Chapter 4, Making Things Move, talks about animating visualizations with d3.js and allowing users to interact with your images.

Chapter 5, Layouts – d3’s Black Magic, explains how d3.js layouts work and shows you how to use the same dataset to get vastly different images. The fancy visualizations out there will no longer look like magic.

Chapter 6, Designing Good Visualizations, looks at a few examples of great visualizations from around the web and discusses just what it is that makes them great.

For More Information:

In this chapter, I’ll show you the basic tools for making simple visualizations in d3.js without going into too much depth so that you can get started immediately. We will go through the basic language of d3.js and also its rules.

We’ll take a stab at creating axes and automatically scaling graphs to fit the viewport, and learn about using Chrome Developer Tools to model our code before going into a full-blown programming bonanza. Through this chapter, we’re going to set up the environment used throughout the book and create an animated chart of a dataset I created from GitHub.

What is d3.js?
The name D3 stands for Data-Driven Documents. Mike Bostock has been openly developing this powerful data visualization library since 2011. It helps you draw beautiful graphics by manipulating data without worrying too much about pixel positions, calculating where things fit on a graph, and so on. If you’ve ever visualized data in Python or similar languages, you’ve probably used something similar to gnuplot. I assure you that d3.js offers a much more pleasurable experience.

The official website, d3js.org, features many great examples that show off the power of d3.js, but understanding them is tricky at best. After finishing this book, you should be able to understand d3.js well enough to figure out the examples. If you want to follow the development of d3.js more closely, the source code is hosted on GitHub at https://github.com/mbostock/d3.
The fine-grained control and its elegance make d3.js one of the most, if not the most, powerful open source visualization libraries out there. This also means that it's not very suitable for simple jobs such as drawing a chart or two—in that case you might want to use a library designed for charting. Many use d3.js internally anyway.

As a data manipulation library, d3.js is based on functional programming principles, which is probably where a lot of the confusion stems from. Unfortunately, functional programming goes beyond the scope of this book, but I'll explain all the relevant bits to make sure everyone's on the same page.

**Setting up a play environment**

D3 combines HTML, CSS, and SVG to create graphics. That means we're going to need an HTML and a JavaScript file. We'll use Chrome Developer Tools to tweak our visualizations and test things out. Let's start with some HTML coding:

```html
<!DOCTYPE html>
<title></title>
<link href="bootstrap/css/bootstrap.min.css" rel="stylesheet">
<div id="graph"></div>
<script src="http://d3js.org/d3.v3.min.js"></script>
<script src="code.js"></script>
```

These six lines of HTML code are the basics we're going to use throughout this book.

The first two lines comprise a minimal HTML5 document. You no longer need to include the `html`, `head`, and `body` tags. Next is the `<link>` tag that pulls in Twitter Bootstrap's CSS rules—a good set of defaults to make things prettier. After that comes the `<div>` tag that will hold our visualization, and finally, there's the `<script>` tag that loads d3.js.

At the end, we include a `code.js` file, where we'll put most of our code. Twitter doesn't offer a hosted version of Bootstrap, so you have to download the whole package from `http://twitter.github.com/bootstrap/` and unpack it next to the other files you're working with. All we need now is a server to run everything. This is because we don't want to get into trouble with browser security models when making Ajax requests. Any server will do, but here's a quick way to get one up and running if you already have Python installed (by default on Mac and Linux).

---

For More Information:

Fire up a console, navigate to your working directory, and run the following command:

```bash
$ python -m SimpleHTTPServer
```

Python will run the SimpleHTTPServer module as a standalone script and create a fully functional local server.

Now point Chrome to `localhost:8000` and fire up the developer console—`Ctrl + Shift + J` for Linux and Windows and `Option + Command + J` for Mac. You should see a blank website and a blank JavaScript console with a command prompt waiting for some code:

![DevTools](image)

**A quick Chrome Developer Tools primer**

Chrome Developer Tools are indispensable in web development. Most modern browsers have something similar, but I thought we’d stick to a single example to keep the book shorter. Feel free to use a different browser.

We are mostly going to use the **Elements** and **Console** tabs: **Elements** to inspect the DOM, and **Console** to play with JavaScript code and look for any problems.

The other six tabs come in handy for large projects. The **Network** tab will let you know how long files are taking to load and helps you inspect the Ajax requests. The **Profiles** tab will help you profile JavaScript for performance. The **Resources** tab is good for inspecting client-side data. Honestly, I have never needed **Timeline** and **Audits** before. One of the favorites from Developer Tools is the CSS inspector at the right-hand side of the **Elements** tab.

For More Information:

It can tell you what CSS rules are affecting the styling of an element, which is very good for hunting rogue rules that are messing things up. You can also edit the CSS and immediately see the results:

A simple histogram

We’ll go through the basics of d3.js by creating a histogram indicating when the GitHub users commit code. We’re going to label axes, make sure things are scalable, and modify animations for that extra bit of flair.

The dataset contains 504,015 repositories and it took me a week to create it out of punchcard data for each repository. A punchcard is just a 7 x 24 grid of buckets, specifying how many commits happened within a specific day and hour. The dataset’s histogram digest is hosted at http://nightowls.swizec.com/data/histogram-hours.json and maps hours to the sum of commits occurring within that hour.

For More Information:
This is what we're aiming for:

![Graph](image)

We begin by taking the environment prepared in the previous section and adding a few lines around the central `<div>` tag:

```html
<div class="container">
  <div class="row">
    <div id="graph" class="span12"></div>
  </div>
</div>
```

The extra `<div>` tags center the graph horizontally and ensure that we have 900 px of width to work with. Don't forget to add the `class="span12"` parameter into the `graph` div. It tells Bootstrap the div should go the whole width of the grid.

To avoid tripping your browser's security restrictions regarding cross-domain requests, you should now take a moment to download the dataset and save it next to the other files. Remember, it's at `http://nightowls.swizec.com/data/histogram-hours.json`.

You can play around with the following code in Chrome Developer Tools to see what it does and then save it in `code.js`. Writing directly to the file also works, but just make sure you refresh frequently. Learning is if you know what each line does.

For More Information:

We begin with some variables as follows:

```javascript
var width = 900, height = 300, pad = 20, left_pad = 100;
```

We’re going to use these to specify the dimensions of our drawing area. The pad variable will define the padding from the edge, with left_pad giving a bigger margin on the left to allow for labels.

Next, we define a horizontal scale, x:

```javascript
var x = d3.scale.ordinal().rangeRoundBands([left_pad, width - pad], 0.1);
```

The x scale is now a function that maps inputs from a yet unknown domain (we don't have the data yet) to a range of values between `left_pad` and `width - pad`, that is, between 100 and 880 with some spacing defined by the 0.1 value. Because it's an ordinal scale, the domain will have to be discrete rather than continuous. `rangeRoundBands` means the range will be split into bands that are guaranteed to be round numbers.

Then, we define another scale named y:

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

Similarly, the y scale is going to map a yet unknown linear domain to a range between `height-pad` and `pad`, that is, 880 and 20. Inverting the range is important because d3.js considers the top of a graph to be y=0.

Now, we define our axes as follows:

```javascript
var xAxis = d3.svg.axis().scale(x).orient("bottom");
var yAxis = d3.svg.axis().scale(y).orient("left");
```

We’ve told each axis what scale to use when placing ticks and which side of the axis to put the labels on. D3 will automatically decide how many ticks to display, where they go, and how to label them.

The last step before loading the data is defining an SVG element for the histogram:

```javascript
var svg = d3.select("#graph").append("svg")
    .attr("width", width).attr("height", height);
```

Switching quickly to the Elements tab, you’ll notice a new HTML element with a width of 900 and a height of 100.
Now the fun begins!

We're going to use d3.js itself to load data remotely and then draw the graph in the callback function. Remember to use Shift + Enter to input multiline code in the Chrome console. Now might be a good time to switch to coding in `code.js` directly and refreshing after every couple of steps:

```javascript
d3.json('histogram-hours.json', function (data) {
});
```

`d3.json` will create an Ajax request to load a JSON file, then parse the received text into a JavaScript object. D3 understands CSV and some other data formats as well, which is kind of awesome if you ask me.

From here on, we put everything in that callback function (before the `}`); bit). Our data will be in the `data` variable. D3 is a functional data-munging library, so we need to transform our dictionary data into a list of simple objects. We do this using the following code:

```javascript
data = d3.keys(data).map(function (key) {
   return {bucket: Number(key),
           N: data[key]};
});
```

`d3.keys` returns a list of keys in the data dictionary, which we then `map` over with an iterator function that returns a simple dictionary for every item. It tells us where an item fits in the histogram (`bucket`) and what value it holds (`N`).

We've turned our data into a list of two-value dictionaries.

Remember the `x` and `y` scales from before? We can finally give them a domain and make them useful:

```javascript
x.domain(data.map(function (d) { return d.bucket; }));
y.domain([0, d3.max(data, function (d) { return d.N; })]);
```

Since most d3.js elements are objects and functions at the same time, we can change the internal state of both scales without assigning the result to anything. The domain of `x` is a list of discrete values. The domain of `y` is a range from 0 to `d3.max` of our dataset—the largest value.

Now we're going to draw the axes on our graph:

```javascript
svg.append("g")
  .attr("class", "axis")
  .attr("transform", "translate(0, "+(height-pad)+")")
  .call(xAxis);  
```
Getting Started with d3.js

We've appended an element called \( g \) to the graph, given it the CSS class "axis", and moved the element to a place at the bottom-left of the graph with the \texttt{transform} attribute.

Finally, we call the \texttt{xAxis} function and let d3.js handle the rest.

Drawing the other axis works exactly the same, but with different arguments:

\begin{verbatim}
svg.append("g")
  .attr("class", "axis")
  .attr("transform", "translate(+(left\_pad-pad)+", 0")
  .call(yAxis);
\end{verbatim}

Now that our graph is labeled, it's finally time to draw some data:

\begin{verbatim}
svg.selectAll('rect')
data(data)
.enter()
.append('rect')
.attr('class', 'bar')
.attr('x', function (d) { return x(d.bucket); })
.attr('width', x.rangeBand())
.attr('y', function (d) { return y(d.N); })
.attr('height', function (d) { return height-pad - y(d.N); });
\end{verbatim}

Okay, there's plenty going on here, but this code is saying something very simple: for all rectangles (\texttt{rect}) in the graph, load our data, go through it, and for each item append a \texttt{rect} and then define some attributes.

The \texttt{x} scale helps us calculate the horizontal positions and \texttt{rangeBand} gives the width of the bar. The \texttt{y} scale calculates vertical positions and we manually get the height of each bar from \texttt{y} to the bottom. Note that whenever we needed a different value for every element, we defined an attribute as a function (\texttt{x}, \texttt{y}, and \texttt{height}); otherwise, we defined it as a value (\texttt{width}).

Keep this in mind when you're tinkering.

Let's add some flourish and make each bar grow out of the horizontal axis.

Time to dip our toes into animations!

Add five lines to the preceding code:

\begin{verbatim}
svg.selectAll('rect')
data(data)
.enter()
.append('rect')
\end{verbatim}

For More Information:  
The difference is that we statically put all bars at the bottom (height-pad) and then entered a transition with .transition(). From here on, we define the transition we want.

First, we wanted each bar's transition delayed by 20 milliseconds using d.bucket*20. This gives the histogram a neat effect, gradually appearing from left to right instead of jumping up at once. Next, we said we wanted each animation to last just shy of a second with .duration(800). In the end, we defined the final values for the animated attributes—y and height are the same as in previous code—and d3.js is going to take care of the rest.

Refresh the page and voila! A beautiful histogram appears as shown in the following screenshot:

Hmm, not really. We need some CSS to make everything look perfect.

Remember that you can look at the full code on GitHub at https://github.com/Swizec/d3.js-book-examples/tree/master/ch1 if you didn't get something similar to the preceding screenshot.
Getting Started with d3.js

Let’s go into our HTML file and add some CSS on line 4, right after including bootstrap:

```html
<style>
    .axis path, .axis line {
        fill: none;
        stroke: #eee;
        shape-rendering: crispEdges;
    }

    .axis text {
        font-size: 11px;
    }

    .bar {
        fill: steelblue;
    }
</style>
```

This is why we added all those classes to shapes. We made the axes thin, gave them a light-gray color, and used a relatively small font for the labels. The bars should be steel blue. Refresh the page now and the histogram is beautiful:

![Histogram Image]

I suggest playing around with the values for width, height, left_pad, and pad to get a feel of the power of d3.js. You’ll notice everything scales and adjusts to any size without having to change the other code. Marvelous!
Summary

We've learned what d3.js is and took a glance at the core philosophy behind how it works. We've also set up a quick and easy environment for prototyping ideas and playing with visualizations. This environment will be assumed throughout the book.

We've also gone through a simple example and created an animated histogram using some of the basics of d3.js. We found out about scales and axes, that the vertical axis is inverted, that any property defined as a function is recalculated for every data point, and that we use a combination of CSS and SVG to make things beautiful.

Most of all, this chapter has given you the basic tools so that you can start playing with d3.js on your own. Tinkering is your friend.
Where to buy this book

You can buy Data Visualization with d3.js from the Packt Publishing website:

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.