Community Experience Distilled

Social Data Visualization with HTML5 and JavaScript

Leverage the power of HTML5 and JavaScript to build compelling visualizations of social data from Twitter, Facebook, and more

Simon Timms

[PACKT] open source*
PUBLISHING
community experience distilled

Social Data Visualization with HTML5 and JavaScript

Leverage the power of HTML5 and JavaScript to build compelling visualizations of social data from Twitter, Facebook, and more

Simon Timms

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Social Data Visualization with HTML5 and JavaScript

First published: September 2013

Production Reference: 1190913

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-654-2

www.packtpub.com

Cover Image by Ravaji Babu (ravaji_babu@outlook.com)

Credits

Author
Simon Timms

Reviewers
Jonathan Petitcolas
Saurabh Saxena

Acquisition Editor
James Jones

Commissioning Editor
Mohammed Fahad

Technical Editors
Dennis John
Gaurav Thingalaya

Project Coordinator
Amigya Khurana

Proofreaders
Amy Guest
Simran Bhogal
Ameesha Green

Indexer
Tejal R. Soni

Graphics
Ronak Dhruv

Production Coordinator
Adonia Jones

Cover Work
Adonia Jones

About the Author

Simon Timms is a developer who works in the oil and gas industry in Calgary, Alberta. He has a BSc in Computing Science from the University of Alberta and a Masters from Athabasca University. He is interested in distributed systems, visualization, and the acquisition of ice-cream.

This is his first book, but he blogs frequently on diverse topics such as code contracts and cloud computing at `blog.simontimms.com`. He is involved in the local .NET and JavaScript community, and speaks frequently at conferences.

I would like to thank the countless open source developers who have made books like this possible through their selfless creation of tools and technologies , such as d3.js and node.js. These are the thinkers, the makers, and the heroes who make the future possible.

Most of all, I would like to thank my wife and son for their undying support while I spent my evenings writing. Without them, I would be without purpose.

About the Reviewers

Jonathan Petitcolas is a web developer specializing in Symfony2 and node.js development. He graduated from the International Institute of Information Technologies (also known as SUPINFO). He spent his last year of study teaching web technologies (and also C/C++) at several French campuses, and also at the Chinese Shandong University of Science and Technology (SUST).

He contributes regularly to several open-source projects at blogs such as Open Source Aficionado. He also shares all his experience and discoveries about programming through his blog (`www.jonathan-petitcolas.com`), or through social networks (`@Sethpolma` on Twitter).

Saurabh Saxena is working in a startup (Banyan learning) as Web Developer and Data Analyst.

He was awarded with Hindustan Pratibha Samman in 2008. He is a certified ethical hacker and the winner of the Evernote programming competition on Interviewstreet.

He holds a rank among the top 100 programmers on Hacker Rank in India. He loves to experiment with technologies such as JavaScript, HTML5, CSS3, Rails, and node.js. He is an open source contributor to Spree Commerce. He is used to spending the rest of his time on Udacity and Code School to learn about new technologies and tools.

He blogs at `saurabhhack123.blogspot.in` and has a website `www.isocialcode.com`.

I would like thank my mom and dad who continually gave me their support when I needed it the most.

Table of Contents

Preface

The world is an interesting place, about this there can be no doubt. We experience the world through our senses, which gather data to be processed by our brains. Frequently the world is disordered, requiring us to think long and hard to get meaning out of things. To ease this process, data can be transformed into other forms that are simpler to understand.

This book is about molding data into a form which is more understandable. It is about taking some of the richest data sources of our time—social networks—and turning their vast array of data into an understandable format. To that effect, we make use of the latest in HTML and JavaScript.

What this book covers

Chapter 1, *Visualizing Data*, introduces us to a world full of ever growing datasets. It also discusses how this river of data can be navigated using visualizations as our canoe.

Chapter 2, *JavaScript and HTML5 for Visualizations*, looks at new features in HTML and JavaScript that present opportunities for visualizing data. It discusses both canvas and Scalable Vector Graphics.

Chapter 3, *OAuth*, examines the often confusing OAuth technology and shows how it can be used to delegate authority to our application. This is because much of the data on social media sites is private and there might be times when we need to get this data.

Chapter 4, *JavaScript for Visualization*, looks at Raphaël.js and d3.js which are great JavaScript libraries that can reduce the pain involved in building visualizations by hand, which is otherwise a time-consuming and error-prone task.

Chapter 5, *Twitter*, looks at how to retrieve data from Twitter and use it to build a visualization.

Chapter 6, *Stack Overflow*, looks at how to retrieve the data API of the ever popular Stack Overflow, which presents some tantalizing opportunities for visualization that can be used to create an interactive graph.

Chapter 7, *Facebook*, explores the Facebook JavaScript API and how to use it to retrieve data to use as the basis of our next visualization. In my mind, the original and still the largest social media network on the planet is Facebook.

Chapter 8, *Google+*, looks at Google's latest foray into social media and how to retrieve data to create a force-directed graph.

What you need for this book

There are very few tools needed to make use of the examples and code in this book. You'll need to install node.js (`http://node.org`) which is covered in *Chapter 5*, *Twitter*. You'll want to download d3.js (`http://d3js.org`), jQuery (`http://jquery.com`), and Raphael.js (`http://raphaeljs.com/`). All the demos can be viewed in any modern web browser. The code has been tested against Chrome but should work on FireFox, Opera, and even Internet Explorer.

Who this book is for

This book is for anybody who is excited about data and wants to share that excitement with others. Anybody who is interested in data that can be extracted from social networks would also find this book interesting. Readers should have a working knowledge of both JavaScript and HTML. jQuery is used numerous times throughout the book, so readers would do well to be familiar with the basics of that library. Some exposure to node.js would be helpful but not necessary.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Once permissions are assigned, Facebook will redirect the user back to your `redirect_uri` allowing you to leverage the token to query the Facebook API."

A block of code is set as follows:

```
OAuth.initialize('<Your Public key>');
OAuth.redirect('facebook', "callback/url");
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
function onSignInCallback(authResult) {
      gapi.client.load('plus','v1', function(){
        if (authResult['access_token']) {
          $('#gConnect').hide();
          retrieveFriends();

        } else if (authResult['error']) {
          console.log('There was an error: ' + authResult['error']);
          $('#gConnect').show();
        }
        console.log('authResult', authResult);
      });
    }
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "This can be done from the **API Access** tab in which you should click on **Create an OAuth 2.0 client ID...**."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1 Visualizing Data

A scant few years ago this book would not have been possible. The rapid expansion in social media, data processing, and web technologies has enabled a fusion of divergent fields. From this fusion we can create fascinating displays of data about exotic topics. The beauty that is inherited in data can be exposed in a fashion that is accessible to the masses. Visualizations such as the following word map (`http://gigaom.com/2013/07/19/the-week-in-big-data-on-twitter-visualized/`), can unlock hidden information while delighting users with an extraordinary experience:

Berkeley
online
degree
Unix
commands
Information
Nate
Useful
new
UC
amp
students
Want
Master
fork
line
000
Big
never
Becoming
master
http
working
DataScience
60
That'll
Silver
School
@mikeloukides
Analytics
master's
bigdata
@kdnuggets
command
@petewarden
University
now
first
Kaggle
@timoreilly
60K
Next
BigData
job
trust
looking
scientists
Program
Cost
program
@gigaom

The size of words in this visualization gives a hint as to their frequency of use. The placement of words is calculated by an algorithm designed to create a pleasing visualization.

In this chapter we'll be looking at how the growth in data is so great that we need to change our tools for looking at it.

There's a lot of data out there

It shouldn't come as a surprise to anybody that the amount of data humans are recording is growing at an amazing rate. Every few years the data storage company EMC produces a report on just how much data is being preserved (`http://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf`). In 2012, it was estimated that between 2005 and 2020 the amount of data stored globally will grow from 130 to 40, 000 exabytes. That works out at 5.2 terabytes for each person on the planet. It is such a staggering amount of information that understanding how much of it exists is difficult. By 2020, it will work out to 11 spindles of 100 DVDs per person. If we switch to Blu-ray discs, which have a capacity of 50 GB, the stack of them required to store all 40, 000 Exabytes would still reach far beyond the orbit of the moon.

The growth in data is inevitable as people put more of their lives online. The adoption of smartphones has turned everybody into a photographer. **Instagram**, a popular image sharing site, gathers some 40 million photos a day. One wonders how many photos of people's meals the world really needs. In the past few months there has been an explosion of video clip sharing sites such as Vine and Instagram, which generate massive amounts of data. A myriad of devices are being created to extend the reach of smartphones beyond gathering photographic data. The latest generation of smartphones include temperature, humidity, and pressure sensors in addition to the commonplace GPS, gyroscopic, geomagnetic, and acceleration sensors. These allow for recording an accurate representation of the world around the user.

An increase in the number of sensors is not a trend that is limited to smartphones. The price of sensors and radios has reached a tipping point where it is economical to create standalone devices that record and transmit data about the world. There was a time when building an array of temperature sensors that report back to a central device was the realm of large SCADA systems. One of my first jobs was testing a collection of IP-enabled monitoring devices at a refinery. At the time, the network hardware alone was worth millions. That same system can be built for a few hundred dollars now. A trip to a crowdsourcing site such as Kickstarter or Indiegogo will find countless Bluetooth or Wi-Fi enabled sensor devices. These devices may find your lost keys or tell you when to water your tomatoes. A huge number of them exist, which suggests that we're entering into an age of autonomous devices reporting about the world. A sort of Internet of things is emerging.

At the same time, the cost per gigabyte of storing data is decreasing. Cheaper storage makes it economical to track data that would have previously been thrown away. In the 1970s, BBC had a policy of destroying recordings of TV programs once they reached a certain age. This resulted in the loss of more than a hundred episodes of the cult classic *Doctor Who*. The low data density of storage media available in the 1960s meant that retaining complete archives was cost-prohibitive. Such deletion now would be unimaginable as the cost of storing video has dropped substantially. The cost for storing a gigabyte of information on Amazon's servers is on the order of a penny-a-month and can be even cheaper if the right expertise are available in house. The Parkinson's law states the following:

> *Work expands so as to fill the time available for its completion.*

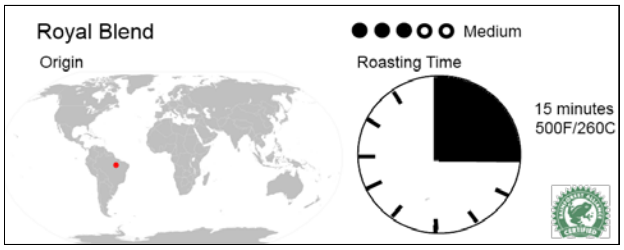
In a restatement of this law, in our case, it would be "*the amount of data will grow to fill the space available to it.*"

The growth in data has made our lives more difficult. While the amount of data has been growing, our ability to understand them has remained more or less stagnant. The tools available to refine and process large quantities of data have not kept pace. Running simple queries against gigabytes of data is a time-consuming process. Queries such as "list all the tweets that contain the word 'Pepsi'" cannot be realistically completed on anything but a cluster of machines working in parallel. Even when the result is returned, the number of matching records is too large to be processed by a single person or even a team of people.

The term "Big Data" is commonly used to describe the sorts of very large datasets that are becoming more common. Like most terms that have become marketing terms, Big Data is defined differently by different people and companies. In this book we'll think of it as any quantity where running simple queries using traditional database tools on consumer grade hardware is difficult due to computational, storage, or retrieval limits.

Understanding the world of Big Data is a complex proposition. Visualizing data in a meaningful way is going to be one of the great problems of the coming decade. What's more, is that it is going to be a problem that will need to be addressed in domains that have not been traditionally data-rich.

Consider a coffee shop; this is not a company that one would expect would produce a great deal of data. However, consumers who are hungry for data are starting to demand to know from whence the beans for their favorite coffee came, for how long they were roasted, and how they were brewed. A similar program called **ThisFish** already exists that allows consumers to track the origin of their seafood (`http://thisfish.info`) all the way back to when it was caught. Providing data about its coffee in an easily accessible form becomes a selling feature for the coffee shop. The following screenshot shows a typical label from a coffee shop showing the source of the beans, roasting time, and organic certification:



People are very interested in data, especially data about their habits. But as interested as people are in data, nobody wants to trawl through an Excel file. They would like to see data presented to them in an accessible and fun way.

Getting excited about data

The truth is that data is interesting! It's amazingly interesting because it tells a story. The issue is that most of the time that story is hidden behind a raft of seemingly uninteresting numbers. It takes some skill to extract the key data and display it to people in a meaningful way. Humans are visual creatures and are more readily able to process images than tables of numbers.

The best data visualizations arise from a sense of passion in the subject of your visualization. Don't we all work better if the subject of our work is something in which we're really interested? Great visualizations don't just educate their viewers, they delight their users. They present data in a novel way that is still easily understood by the audience. Great visualizations strip away the excess information to reveal a kernel of information. At the same time, great visualizations have a degree of beauty to them. Don't be fooled into thinking that this beauty serves no purpose. In a world of ever shortening attention spans, there is still a place for beauty. We still stop and pause for a moment when presented with as aesthetically pleasing visualization. The extra few seconds that the beauty buys you may be what keep people interested long enough to take in your meaning.

Even the most benign data has a story worth telling. To most, there is very little that seems less interesting than tax revenue statistics. However, there have been some very compelling stories found within that raft of data. The data tells a story about which companies are avoiding paying tax revenues. It tells another story about which cities have the highest per capita income. Within that boring data are countless interesting stories that can be extracted though a passionate application of data visualization.

Data is a lot of things, but it is never boring. You can get excited about data too and uncover the hidden stories in any dataset. In every dataset, there is an interesting conclusion waiting to be exposed by a data sleuth such as yourself. You should share your excitement with others in the form of data visualizations.

Data beyond Excel

By far, the most popular data manipulation and visualization tool in the world is Microsoft Excel. Excel has been around for almost three decades, and during that time has grown to be the de facto tool on which businesses rely to perform data analytics. Excel has the ability to sort and group data and to create graphs for the resulting information.

As we saw previously, the amount of data in the world is huge. The first step in most data visualizations is to filter and aggregate the data down into a dataset that contains the key insights you want to share with your users. If it sounds like extracting, meaning that it is an opinionated process, that's because it is. Presenting an unbiased visualization is just about impossible. That's okay, though. Not everybody is an expert on your data, and guiding others to your conclusions is valuable.

You'll find that the data you have from which to derive visualizations is hardly ever in a format you can use right off the bat. You will need to manipulate the data to get it into a form you can use. If your source dataset is small enough and your manipulations sufficiently trivial, you may be able to do your preprocessing in Microsoft Excel. Excel provides a suite of tools for sorting, filtering, and summarizing data. There are numerous books and articles available on how to work with data in Excel as well as how to create graphs, but we won't delve into it here.

The problem with Excel is that it is old news. Everybody has seen the rather pedestrian graphs you get out of Excel. With the exception of a couple, these are the same charts which were produced by Excel 95. Where is the excitement about data? It seems to be missing. If you create your visualizations wholly in Excel, your users are going to miss out on your enthusiasm for data.

Swiss army knives are famous for having a dozen different features. You can use the same tool to open a bottle of wine as you use for removing stones from the shoes of horses (a far more common application around most parts). When you build a tool to be multi-functional, you end up with a tool that does nothing particularly well. Simply looking at the length of the help index for Microsoft Excel should tell you that Excel falls solidly into the category of multi-functional tools. You can do your accounting with Excel or track how quickly you can run a 5K; you can even build graphs with that data. But what you can't do is build really good graphs. For that, you're going to need specialized tools with a narrower focus on data visualizations.

Social media data

We've talked a lot about visualizations and data, but the other part of this book's title is to do with social media. Unless you've been living in a cave without Internet, you will be at least slightly aware of the social media wave that has swept the planet in the last decade. Has it really been only a decade? Facebook was founded in 2004. While one can point to examples from before 2004, I would argue that Facebook was the first social media site to enter into the common consciousness of the population.

Defining what exactly makes a site a social media site is difficult. There needs to be some aspect of social interaction on the site and some sort of a connection between the users. To avoid labeling any site with a comment section as a social media site, the primary purpose of the site must be to enable the interactions between users. Content on these sites is typically user-generated rather than being created by the owners of the site. Social media sites enable interaction between users with similar interests.

Why should I care?

The role that social media now plays in our world cannot be understated. Even if you avoid membership in all social media sites and believe that social media has no impact upon your life, it does. A great example of the real-world impact of social media is how reliant news media has become on social media. Earlier this year, the Associated Press' Twitter account was hijacked and several messages were sent suggesting that the White House had been attacked by terrorists. While the news was quickly rebuffed, stock markets declined sharply on the news. Had the subterfuge not been so quickly discovered, the real-world consequences could have been far worse.

Data from social media provides a context for events happening around the world. One has simply to look at the trends on Twitter to pick out the important news stories of the day. As newspaper subscriptions drop, the number of people on Twitter, Facebook, and other sites grow. Traditional news outlets have started to integrate commenting and sharing on their stories via social media. The commentary on the story often becomes the story instead of simply providing a meta-story. Many commentators have pointed to the importance of Twitter in the Arab Spring and even in protests in the US. Social media is quickly overtaking more traditional sources of news and is becoming a driving force for society.

Social media is not limited to person-to-person interactions. More and more, it is being used by businesses to connect with their customers. Frequently, the best way to get service from a faceless corporation is to post a message on their Facebook page or send them a tweet. I have certainly had the experience of tweeting about a company or a service only to have their social media people reach out to me. Anything that empowers companies to develop better relations with their customers is a powerful tool and likely to have a long life.

From the perspective of visualizations, social media is a phenomenal source of interesting data. I can think of very little that is as compelling as a source of data as the social interactions between people. Humans evolved to be social animals so we have a built-in interest in what is happening in our social circles. In addition to their websites, many social media sites have APIs that promote building applications that use their data. The theory is that if they can enable an ecosystem for their valuable data, people are far more likely to visit their site frequently and third-party applications may even draw in new users.

Social media is the very definition of Big Data. Facebook has something like a billion users, each of which may generate a dozen pieces of data a day. Twitter, LinkedIn, and Facebook have all created their own database technologies after having found the amount of data with which they have to deal, to be too large for traditional databases. Fortunately, there is little need to work with the full scale of social data. Narrower sets of data can be accessed through the various data access APIs. The key is to shift as much of the filtering and aggregation to the social media sites as possible. By exploring the available information, it is possible to draw interesting conclusions and expose information through visualizations that aren't typically apparent to users.

HTML visualizations

The final piece in the puzzle is HTML5. When I was young, a new version of HTML meant another long-winded specification from the World Wide Web consortium. The specification process for a new version of HTML would take several years and would be planned out by a committee with members from large technical organizations such as Microsoft and IBM. While there is an HTML5 specification, it is not as formal as previous iterations. The term HTML5 has come to describe a collection of future-oriented technologies that can be used to create powerful web applications.

HTML5 includes specification for diverse features such as the following:

- Web workers (multi-threaded JavaScript)
- Touch events for touchscreen devices
- Micro-data formats
- Canvas
- Scalable Vector Graphics
- Camera API
- Geolocation API
- Offline data

Through these new APIs and features, HTML5 has become a major player not just on browsers, but also on mobile devices and on the desktop. Through toolkits such as PhoneGap (`http://phonegap.com/`), HTML and Microsoft's WinJS JavaScript can be used as primary development languages on iPhones, Android, Windows Phone, and even Blackberry. The native APIs are bound to JavaScript equivalents opening up the camera, GPS, and filesystem to JavaScript applications. HTML5 can also be used as a development platform for Windows 8-style applications (previously known as Metro). On non-Windows platforms, desktop applications can be developed in HTML5 using a toolkit-like Adobe Air (`http://www.adobe.com/products/air.html`). HTML5 offers a multi-platform development environment that allows taking skills from the Web to tablet to desktop.

The offline data tools remove the dependence on having a web server to serve content to your application. Embedding data directly on the client machine instead of having to pull it down repeatedly from a server allows for applications to be truly mobile—the network is no longer crucial.

HTML5 has been hugely beneficial to visualization developers. Canvas and SVG both offer enticing functionalities. CSS3 also allows for a greater degree of flexibility around styling. Before HTML5 came onto the scene, interactive data visualizations in a browser could best be achieved using third-party tools such as Java Applets or Adobe Flash. The adoption rates for these technologies, while high, still cut off a large number of users. Even with high adoption rates, the versions of these tools being run in the wild were frequently archaic. Neither Java Applets, nor Adobe Flash is available on the increasingly popular mobile platforms. HTML5, on the other hand, is now supported in some form on the vast majority of smart phones.

One of the best features of developing a visualization in HTML is that it is possible to allow users to interact with the visualization. Famous visualizations such as the *London Underground Map* have been crippled by being drawn on a static piece of paper. Interactions provide a whole new level of user engagement—previously impossible. It should not surprise you if users of interactive visualizations find ways to manipulate the visualization to derive whole new conclusions.

The industry support for HTML and JavaScript technologies is impressive. All the technology giants have invested heavily in developing browsers and development tools based on HTML5 and JavaScript. The pace of change in the web development sphere is stunning. There is not a week that goes by when I fail to hear about an innovative new JavaScript library or a new take on a development platform. The ready availability of cloud-based hosting has enabled startups to flourish on the web.

When choosing a tool in which to develop visualizations, HTML provides an excellent option. Broad support, good tooling, and a well-known API ensure that developing will be a pleasure. Well, maybe not a pleasure, but at least relatively painless. HTML and JavaScript are the lingua franca for all web developers. No matter if development is being done with Ruby on Rails, ASP.NET, or even Wordpress as a backend, the frontend is always going to be written in HTML and JavaScript. This gives a big pool of developers from which talent may be pulled.

Summary

Communicating information to users is tricky. The problem is compounded by the huge quantity of data that is now available at the click of a mouse or the punch of a key. As a visualization developer, it is your role to sort through clouds of irrelevant data to extract the bits in which you're interested and then to present that data to your users in an interesting way. People are interested in data, but they are rarely interested in sorting through reams of tabular data. Visualizations are frequently the best tools for presenting that data to your users.

The confluence of readily accessible, high quality, social data from social media sites coupled with new visualization tools present a never before seen opportunity to create interesting visualizations. Through the passion of developers who can see beyond the standard Microsoft Excel graphs and tables, there is a future for not just static visualizations but also interactive, fun visualizations that will delight users while they explore previously invisible aspects of data.

In the next chapter we'll examine some of the ways in which we can create visualizations using modern web development tools.

2 JavaScript and HTML5 for Visualizations

In the previous chapter I mentioned that there have been some developments in HTML5 that have made visualizations far easier. This chapter is going to explore a couple of them, namely the following:

- Canvas
- Scalable Vector Graphics (SVG)

If you're familiar with the functionality of these two tools, you might want to give this chapter a skip.

Canvas

Canvas is one of the features of HTML5 that gets the most play in technical articles and in demos. The things that can be done with canvas are very impressive, so it is no surprise that it appears so frequently. Canvas provides a low-level bitmap interface for drawing. You can think of it as Microsoft Paint in a browser. The images that are generated on the canvas are all raster images, meaning that the images are built from a grid of pixels rather than a set of geometric objects, as is the case in a vector image. Interaction with elements drawn on a canvas must be done through filters and global transformations; precise control is problematic if not impossible.

Creating a canvas element on your page is simple. You simply need to add an HTML element such as the following:

```
<canvas width="200" height="200">
Alternate text here
</canvas>
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can also find all the code examples for this book at `https://github.com/stimms/SocialDataVisualizations`.

This will create a square canvas of 200 x 200 pixels. Should the browser being used not support the canvas element, the alternate text will be shown. Fortunately, it is pretty rare that you'll see this warning, as canvas has wide support. By the middle of 2013 canvas was available to 87 percent of Internet users. For the latest numbers, check out `http://caniuse.com/#feat=canvas`. This site also has browser support information for other HTML5 features. There is even support for mobile browsers on iOS, Android, BlackBerry, and Windows Phone. The only commonly used browsers that don't have support for canvas are versions of Internet Explorer prior to Version 9.

If you're in a situation where your target demographic makes heavy use of older Internet Explorer versions, all is not lost. There is a handy JavaScript **polyfill**; a downloadable piece of code that provides canvas functionality to older versions of Internet Explorer. The library is available at `https://code.google.com/p/explorercanvas`. To make use of it, a conditional include may be used, as shown in the following code snippet:

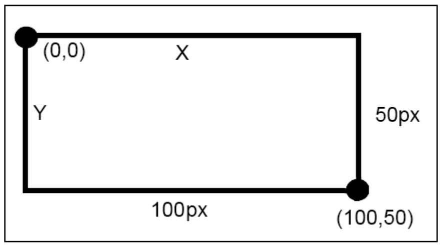
```
<head>
<!--[if lt IE 9]><script src="excanvas.js"></script><![endif]-->
…
</head>
```

This will only include `excanvas.js` if the browser running is an older version of Internet Explorer.

Excanvas does not fully support canvas, with features such as shadows and 3D missing, but the vast majority of the features are available. There is also a performance hit to using the JavaScript version of canvas, so, if you make use of animations, they may not be as smooth as they would on a modern browser. This is a small price to pay for reaching the remaining few percent of users. As browsers are updated, this problem will become less and less important.

Drawing simple shapes on the canvas is easy, but there is also the power to draw some very complex objects, including drawing in 3D. We'll limit our discussion to some of the simpler shapes and functions that would be useful in creating visualizations.

Before we get into drawing, we hold the introduction of the coordinate system for canvas. The origin of the coordinate system is located in the top-left corner and grows towards the bottom-right. To draw on the canvas, as shown in the following figure, JavaScript is used:



The first step in drawing is to get a handle for the instance of the canvas to which you would like to draw. You can, of course, have as many canvas elements on a page as you wish. Here, we'll create a canvas with an ID of `demo`:

```
<canvas height="200" width="200" id="demo">
</canvas>
```

We can now select the element using standard JavaScript methods:

```
var demoCanvas = document.getElementById("demo");
```

Alternately, if you're making use of a CSS selector library, such as jQuery, you can select the element using that:

```
var demoCanvas = $("#demo")[0];
```

The next step is to get a reference to the drawing context itself. The context contains a collection of methods for drawing and will be what we use for all canvas operations:

```
var context = demoCanvas.getContext("2d");
```

Canvas supports a number of basic shapes from which it is possible to build complex objects. The simplest shape is the lowly rectangle. This can be created by calling the `strokeRect()` function:

```
context.strokeRect(20, 30, 100, 50);
```

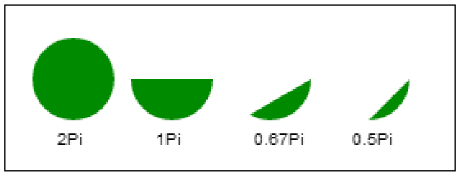
This will create a rectangle that starts at coordinates (20, 30) with a width of 100 pixels and a height of 50 pixels. In fact, this will draw the previously shown rectangle but shifted 20 pixels to the right and 30 pixels down. So, the signature for this method is to give the x, y coordinates for the starting point of the rectangle followed by the width and height. In addition to the `strokeRect()` function, there are `fillRect` and `clearRect`. When drawing on canvas, one can draw a stroke which is a line, draw a filled structure, or clear the content of an area.

If rectangles aren't your style, perhaps you would be more interested in drawing a circle? Canvas actually considers a circle to be a special case of an arc. Thus, to draw a circle with a radius of 50px, you need to specify not just the center point and radius, but also the starting and ending angle. The code for such a circle is as follows:

```
context.arc(75, 75, 50, 0, 2*Math.PI);
context.fill();
```

Here, the center point is given as (75, 75) and the radius 50. The fourth term is the starting angle which we give as 0 and the `2*Math.PI` is the ending angle—the whole way round the circle. An optional final argument determines if the arc is to be drawn anti-clockwise. It defaults to `false` or `clockwise`.

Alerting the final parameter gives different arcs, as shown in the following screenshot:



As you can see, all of the angles used in canvas are denoted in radians. To convert from an angle in degrees to one in radians, you can multiply it by `Pi/180`.

For more complex shapes, canvas supports straight line or a path. To draw a line, you set a starting coordinate, then give an ending point, and then call the `stroke()` function. Canvas will extrapolate where to draw the line, as shown in the following code snippet:

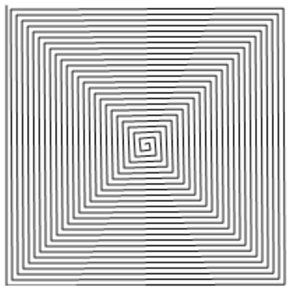
```
context.beginPath();
context.moveTo(0,0);
context.lineTo(50,120);
context.closePath();
context.stroke();
```

Here we start a new path; the pen is moved to (0, 0), then a line is drawn to (50, 120), and then the path is ended. It is important to begin and end your paths, or subsequent calls to `stroke()` functions will result in a continuation from the last point. You can think of it like using a pen; `beginPath` puts the pen on the piece of paper, `moveTo` temporarily lifts the pen and moves it, `lineTo` moves the pen to the destination and draws a line behind it, and finally `closePath` picks the pen back up off the paper. Without picking the pen up, the next time you draw a line to somewhere, the pen will already be on the paper and you'll get an extra line.

If you think the syntax for drawing a line is a bit arcane, you're not alone. The multiple calls enable you to build more complex lines with several segments. By using a loop, we can build relatively complex shapes, as shown in the following code snippet:

```
var width = 200;
var height = 200;
context.moveTo(0, 0);
for(i = 0; i< 100; I += 4)
{
   context.lineTo(i, height-i);
   context.lineTo(width - 1-i, height - 1 -i);
   context.lineTo(width - 2-i, i+2);
   context.lineTo(i+3, i+3);
}
context.stroke();
```

This code will generate a spiral. On each iteration of the loop, we move towards the center of the image by 4 pixels, one pixel on each edge. The result is as shown in the following screenshot:



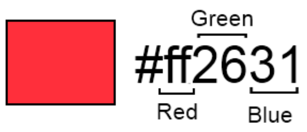
The canvas maintains a state between calls to it, so, if you set the color of the fill using `context.fillStyle`, all subsequent calls to fill a shape will take on the same fill style. As you can probably imagine, this is a common source of bugs when building a visualization using canvas. It is especially problematic when calling functions to operate on the canvas.

Fortunately, there is an easy solution: the context state can be saved and restored when entering and exiting a function. It is polite to keep your functions from messing around with a global state and will certainly reduce the number of bugs:

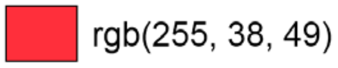
```
function fillCircle(context, x, y, radius){
   context.save();
   context.fillStyle = "orange";
   context.beginPath();
   context.arc(x, y, radius, 0, 2* Math.PI);
   context.fill();
   context.restore();
}
```

In the first line of the function, the current drawing context is pushed onto a stack and then it is restored on the last line of the function. By doing this, we can make as many changes to the context within the method as we like and rest assured that the context of the caller will not be corrupted.

Canvas has support for a full palette of colors and also allows for transparency and even gradients. So far, the examples have used color names. These names date way back to the mid 1990s and actually hail from the X11 windowing system. However, there are two other ways to specify a color for canvas. The first is by using a hexadecimal string that specifies the values for red, green, and blue as two-digit hexadecimals digits. The larger the value, the higher is the intensity of that color, as shown in the following figure:



The final, and my preference, is to use decimal RGB values, as shown in the following figure, which I think is far more readable and also easier to build programmatically:



A slight variant on the decimal format is to use the `rgba` function instead of `rgb`. This adds an extra parameter, which is a decimal number between 0 and 1, which denotes the opacity. 1 is fully opaque and 0 is completely transparent.

As canvas is a raster-based drawing system, it is possible to include most other raster files in it. Raster file formats include JPEG, PNG, GIF, and BMP. Being able to import an existing image can be a very handy tool for visualizations.

However, there are some caveats to using images on canvas. When you're including an image, you can't just load it from a URL directly. First, a JavaScript Image object needs to be created and the source of that image needs to be set to the URL. This Image object can then be used on the canvas:

```
Var image = new Image();
image.src = "logo.png";
```

Requesting images from another domain can be tricky. In order to maintain security in the browser requesting data, other domains are restricted. For images, you can request permission from the hosting domain to use the image by setting the `crossOrigin` property on the image:

```
Var image = new Image();
image.crossOrigin = "anonymous";
image.src = "http://codinghorror.typepad.com/.a/6a0120a85dcdae970b0128
776ff992970c-pi";
```

Here, the value of the `crossOrigin` policy has been set to `anonymous`. This means that the browser won't pass on any authentication information to the server hosting the image. You can also set a value of user-credentials if you do want to pass credential information. Support for `crossOrigin` on images is relatively new, so you may do better to host the images on the same domain as the canvas.

Because loading an image can take some time, it is not advisable to set the value of `src` on an image and immediately attempt to draw it on the canvas. Instead, you should hook into the `onload` event on the image:

```
function drawImage()
{
   var img = new Image();
   img.src = 'worksonmymachine.png';
   img.onload = function(){
      var context = document.getElementById('example').
getContext('2d');
      context.drawImage(img, 0, 0);
   }
}
```

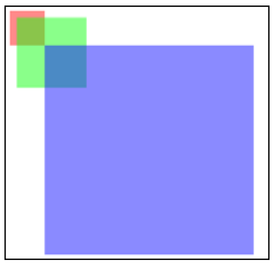
Using the `onload` function will prevent an empty image from being rendered to the canvas. If you're loading multiple images, the order in which they are loaded is indeterminate. You may wish to check that all images are loaded before continuing. Complex dependency chains can be managed using jQuery's `Deferred` functionality. The second and third parameter to `drawImage` is, predictably, the coordinates at which to draw the image. There are more advanced versions of `drawImage` that allow for scaling and cropping of an image before drawing it to the canvas.

The final feature of canvas which we'll look at is the transformation. When composing more complex scenes or visualizations, it is frequently easier to build the object at a different scale, location, or orientation. Transformations provide a mechanism for altering the shapes you draw on canvas.

A function, in this context the scaling function, will multiply every coordinate by the x or y scaling factor. Canvas provides for independently scaling the x and y values. This means that it is easy to stretch a shape in just one direction:

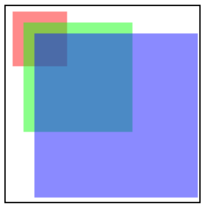
```
var colours = ["rgba(255,0,0,.5)", "rgba(0,255,0,.5)",
"rgba(0,0,255,.5)"];
for(var i = 0; i< 3; i++)
{
    context.fillStyle = colours[i];
    context.scale(i+1,i+1);
    context.fillRect(10, 10, 50, 50);
}
```

The preceding code, which simply draw a series of three rectangles, will produce an output that looks like the following screenshot:



You'll note that our scaling didn't just create larger images, it also multiplied the coordinates of the rectangles. The other thing to note is that the size increase doesn't seem to be uniform. That's because canvas is stateful. If you apply several scale functions in a series without resetting the scaling, each one will build on the last scaling. You can keep a track of your transformations and apply a reversing function. For instance, if you have scaled by 2, you can scale by the multiplicative inverse of 2, which is 1/2, to get back to the original scaling. It is more likely easier to save the context and restore it using the `save` and `restore` functions we spoke of earlier.

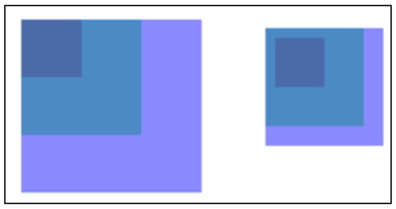
If we modify our function, you can see that the resulting image is very different, as shown in the following screenshot:



Frequently, when applying a scaling transformation, you'll want to apply a translation transformation to move the starting coordinates to where you expect them to be. You can do this by employing a translate transformation:

```
var x = y =10;
var width = height = 100;
for(var i = 2; i>= 0; i--)
{
   var scalingFactor = i+1;
   context.save();
   context.fillStyle = colours[i];
   context.translate(x * -i, y *-i);
   context.scale(scalingFactor, scalingFactor);
   context.fillRect(x, y, width, height);
   context.restore();
}
```

On the highlighted line, we shift the canvas over so it is as if the rectangle is drawn at the origin. The order of the application of transformations is important, as shown in the following screenshot:



On the left-hand side, you can see the output of our code and on the right-hand side, the result of swapping the `translate` and `scale` operations.

There are a lot of other great features of canvas. Going into everything is just too much for a book of this length; it is most likely a book in its own right.

Scalable Vector Graphics

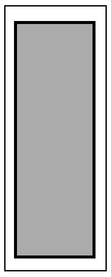
Scalable Vector Graphics, or as they are more commonly known, SVGs, are another feature relatively new to HTML. They fulfill a similar role to canvas, but differ by the fact that they are vector-based instead of raster-based. This means that every image is made up of a series of basic shapes. This might sound a bit like canvas, after all, we created all our canvas images using basic shapes. The difference is that with SVG, the basic shapes remain as distinct objects after they've been drawn. With canvas, the source commands used to create the image are lost as soon as it is rendered. The information about the source of the pixel is lost in a jumble of canvas commands.

Conveniently, SVGs are stored as XML files. While I would typically not even consider the idea of storing anything but the combination to my safety deposit box, in such an inaccessible file format it does integrate nicely with HTML. Raster images are typically linked in a separate file from the HTML. SVGs can be embedded directly into the HTML document. This technique can be used to reduce the number of server trips necessary to render a page on a user agent. However, the real advantage is that it allows for the SVG to be integrated into the HTML **Document Object Model** (**DOM**), allowing you to manipulate the SVG using the same techniques you might use to manipulate any other element.

The source for a simple SVG may look like the following:

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <rect width="50"
        height="150"
        x="20"
        y="20"
        stroke="black"
        stroke-width="2"
        fill="#a3a3a3" />
</svg>
```

You can paste this into any HTML document and you'll get a simple rectangle that looks like the following screenshot:



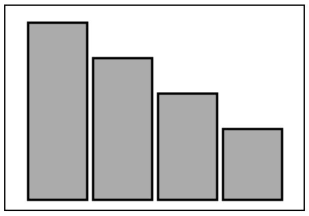
The code is quite easy to understand, and the syntax should be familiar to anybody who has built a website. We first open new SVG element. Without any explicit sizing information the SVG fills its container. Inside of the SVG, we create a new rectangle with a width of 50px and a height of 150px. The outline of the rectangle is black and has a width of 2px while the inside is filled with a grey color.

The primitives you can use to build your images should also be somewhat familiar, now that you've seen canvas in action. `rect` and `path` remain unchanged from canvas. However, SVG differs in its treatment of circles and provides an actual `<circle>` tag as well as an `<elipse>` tag for round shapes with two foci. `<polygon>` and `<polyline>` tags provide for drawing free-form straight-edged shapes with the polygon being a filled shape and the polyline being just a line. Should you desire a more curvy shape, SVG provides a `path` element that allows for defining complex curves and arcs. It is very tricky to build a curved path by hand. Typically, for curved paths you'll want to make use of an editor or an SVG library. Finally, SVG has support for writing text using the aptly named `<text>` element.

Building multiple elements in SVG is as simple as adding another child to the SVG , as shown in the following code snippet:

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <rect width="50" height="150" x="20" y="20" stroke="black" stroke-
width="2" fill="#a3a3a3" />
  <rect width="50" height="120" x="75" y="50" stroke="black" stroke-
width="2" fill="#a3a3a3"/>
  <rect width="50" height="90" x="130" y="80" stroke="black" stroke-
width="2" fill="#a3a3a3"/>
  <rect width="50" height="60" x="185" y="110" stroke="black" stroke-
width="2" fill="#a3a3a3"/>
</svg>
```

This results in an image that looks like the following:



As you can see, the code is quite repetitive. On every single rectangle, we specify the `stroke` and `fill` information. This repetition can be eliminated in two ways. The first is to use a group of elements to define the styling information. SVG provides a generic grouping container which is denoted by the `<g>` tag. The styling information can be applied to that container instead of the individual elements:

```
<g stroke="black" stroke-width="2" fill="#a3a3a3">
  <rect width="50" height="150" x="20" y="20"  />
  <rect width="50" height="120" x="75" y="50" />
  <rect width="50" height="90" x="130" y="80" />
  <rect width="50" height="60" x="185" y="110" />
</g>
```

An alternative is to use CSS to do the styling for you:

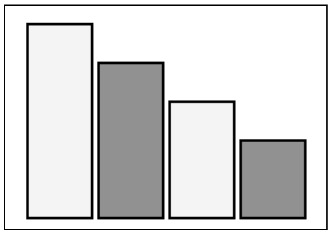
```
<style>
  rect {
    fill: #f3f3f3;
    stroke: black;
    stroke-width: 2;
  }
</style>
<svg xmlns="http://www.w3.org/2000/svg" version="1.1" id="graph1">
  <rect width="50" height="150" x="20" y="20" />
  <rect width="50" height="120" x="75" y="50" />
  <rect width="50" height="90" x="130" y="80" />
  <rect width="50" height="60" x="185" y="110" />
</svg>
```

In the preceding code, the styling information is attached directly to all elements of type `rect`. Typically, you would want to avoid using broad selectors such as these, as they will apply to all the SVG elements on the page. It is better to narrow the selectors either by making them apply to just that one SVG or, preferably, by assigning a class to the SVG elements you wish to style. It is generally preferred to style your elements, even those that are part of an SVG, using CSS. It is likely that your SVG will contain multiple rectangles that you don't want to take on the same styling.

The style properties (`fill, stroke`, and so on) used in the CSS do not differ from those used in in the SVG markup. More advanced CSS selectors are also available, such as `nth-child`, which selects just children matching a specific pattern. Consider the following code snippet:

```
rect:nth-child(even)
{
   fill:#878787;
}
```

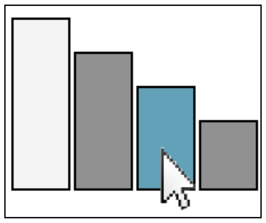
Adding the preceding code in our example will, very simply, create a zebra striping effect on our graph, as shown in the following screenshot:



We can even get fancy and use CSS to add some interaction to our graph by simply specifying a `:hover` pseudo selector in the CSS and changing the color under the cursor:

```
rect:hover
{
   fill: rbg(87,152,176);
}
```

The following screenshot shows the resulting graph:



Of course, having the SVG as part of the DOM opens other doors to factors other than styling. It is also possible to manipulate the elements of an SVG using JavaScript. You can even assign event listeners to the SVG elements.

By making use of the fantastic jQuery library, we can easily add event listeners to the nodes in the graph SVG we've been building so far:

```
$("rect").click(function(){
    alert($(this).attr("height"));
});
```

If you've never seen jQuery before, what is happening here is that we're selecting all the `rect` elements on the page, and when a click is fired, it opens an alert box with the height of the column on which we clicked.

Throughout this book, we'll be making extensive use of the jQuery library and this style of lambda-based programming. If you're not familiar with jQuery, it would be advisable to take a break and read some tutorials, such as `http://try.jquery.com/`.

We've covered all the basic functionality of SVGs, but there is one advanced feature I'd like to mention: filters. **Filters** are transformations that can be applied to elements of an SVG. These filters go beyond the scaling and translation transformations we saw in canvas, although both `scale` and `translate` are supported in SVG. There are about 20 of these filters and each one performs different transformations. We won't be able to go into each, but we'll look at a couple of them.

One of the most common requirements in visualization is to give things a 3D feel. Full 3D can be very difficult, but we can trick the eye by using shadows. These shadows can be created using a combination of three different filters: offset, Gaussian blur, and blend.

To use a filter, we start by defining it. A `filter` element can be defined as a number of sequentially applied filters. To figure out what filters we need for a shadow, we can work backwards from the properties of a shadow. The first thing to notice is that shadows are offset from the things casting them. For this, we can use an offset filter that will shift the element in one direction or another. Where you want to shift the element depends on where your light source is. For our purposes, let's say that it is above and to the left of the SVG. This will cast shadows down and to the right:

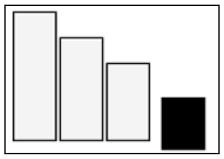
```
<defs>
    <filter id="shadowFilter" width="175%" height="175%">
     <feOffset result="offsetImage" in="SourceAlpha" dx="5" dy="5"/>
    </filter>
</defs>
```

On the filter line, we need to specify a height for our filtered element, which is greater than the original. If we don't do so, much of our shadow will be cut off as it extends beyond the boundary of the source object. Here we have given our filter an ID so it is easily applied later. You'll also note that we've specified an `in` and `out` property for `feOffset`. This allows us to chain the filters together. In our case, we're taking `SourceAlpha`, which is just the alpha, or the `transparency` property from the original image.

Let's apply this filter to just one element of our graph so we can see what is happening to it. I've removed the other styling so as not to confuse matters. The filter is applied by using the `filter` attribute and giving it the ID of the filter created previously:

```
<rect width="50" height="60" x="185" y="110"
filter="url(#shadowFilter)"/>
```

The following will be the result:



Shadows are also fuzzier than the original image. This can be achieved by using a Gaussian blur filter:

```
<feGaussianBlur result="blurredOffset" in="offsetImage"
stdDeviation="8" />
```

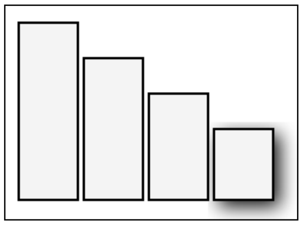
Gaussian filters randomly move points inside your image based on a normal distribution function. You might wish to play around with the standard deviation to achieve different blur effects; I found somewhere in the 8-12 range to be good for shadows:

```
<feBlend in="SourceGraphic" in2="blurOut" mode="normal" />
```

Finally, we want to take the blurred black box we've created and combine it with the original:

```
<feBlend in="SourceGraphic" in2="blurredOffset" mode="normal" />
```

Applying this filter on hover gives a very convincing pop-up effect when users hover over the image, as shown in the following screenshot:



SVG provides for easy manipulation of parts of an image though the styling tools that are already well known to you from your work with CSS. At the same time, being able to attach events to the image allows the creation of impressive user interaction.

Which one to use?

Deciding whether to use canvas or SVG can be a difficult problem. It mostly comes down to which one feels more comfortable. Those with a background in computer graphics or animation are more likely going to be happier with canvas, as the `redraw` loop of canvas will be familiar. Canvas is better suited to redrawing entire scenes or even if you plan on using 3D elements. If your visualization makes use of textures or rendered images, canvas' ability to draw them to the canvas directly is almost certainly going to be advantageous. For visualizations that have some reliance on maintaining a fast frame rate, canvas is generally high performing.

On the other hand, SVG can be a much simpler technology to use. Each element in an SVG can be individually manipulated, which makes small animations far easier. The integration with the DOM allows for events to be fired on interaction with a single element of the SVG. To achieve that in the canvas, you must manually track what is being drawn at that location. That SVG can also be styled using CSS, which allows for components to be more easily reusable on sites with different themes.

For the purposes of this book, we're going to focus on SVGs. The resolution independence of SVG coupled with the ease of use and fantastic support libraries makes it a logical choice. I don't believe there is a visualization we'll be creating that cannot be created with canvas but the effort would be far greater. This is doubly so for the interactive visualizations.

Summary

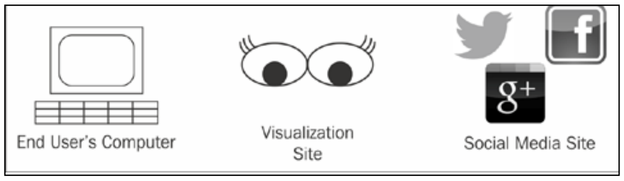
You should now be able to make an informed decision between SVG and canvas-build simple static images. We're going to take a break from the visual aspects in the next chapter and talk about the OAuth protocol, which is used by many social media sites to protect their data.

3
OAuth

Creating visualizations is really only half the battle; the other half is getting high quality data to drive the visualization. There are a lot of potential data sources that you may wish to exploit. Almost every country has a national statistics organization tasked with gathering and analyzing economic and social statistics. In the past few years, many governments have begun adopting Open Data initiatives. Many businesses are also centered on providing usable data; think about the amount of data provided by various stock exchanges. You may even have access to internal company data you're using to drive your visualization, or your visualization may be part of a larger application that will provide you with data.

Another source, and the source we're interested in for this book, is social media. Social media sites have a plethora of data available to their users. The vast majority of social media sites provide APIs for accessing data in a programmatic fashion. Frequently, this data is personalized for your user account. For instance, Twitter filters the tweets you see by the users you're following, and Facebook similarly shows you updates from your friends. Some data is restricted, such that only certain groups of people can see it. You may not want the whole world seeing your Facebook updates, so you set permissions to only allow friends to see it. In order to personalize the data, most social networking sites require authentication and authorization for their APIs.

Typically, the data you want to show users is related to their own social media account. In most cases, your visualization will not be hosted on the same system as the data you're looking to use, as shown in the following figure. What's more important is that the consumer of your visualization, the end user, is likely to be on yet another system:



This means that there needs to be some way to get the social media data from the social media site to your visualization, and then on to the end user. On the surface, there seems to be the following two options:

- Ask the user for their social media password and use that to authenticate and check authentication against the social media API
- Ask the user to retrieve the data required for the visualization and send that on to your visualization

Neither option is particularly desirable. Users are likely to be resistant to giving out their passwords, especially to something as important as their social media sites. You probably don't want to take ownership of their password either, as that is an extra security headache for you. Most users are insufficiently tech-savvy to extract the data needed from their social media sites and send it onto your application. Even those who have the technical skills are not going to be impressed with the amount of work that will entail. Certainly, standard export mechanisms could be established and refined to make it easier for users, but before we develop a system for this, perhaps is there a third way?

The trick seems to be in finding a way to get credential information from the end user to the social media site without telling the intervening visualization site too many details. As with many computer problems, a parallel problem and solution may be found outside of the world of computing.

Many electronic garage door openers and home security systems offer guest codes. These codes may have restrictions on them, such as they only operate on certain days of the week or a certain number of times. The purpose of these accounts is to provide limited access to your house for, say, cleaning people or trade people. A similar concept is purported to exist in high-end automobiles: a valet key causes the car to operate in a restricted mode. A similar mechanism could be used to grant your visualization access to a restricted portion of the social media site.

The OAuth protocol provides a mechanism for authenticating applications to make use of your social media data and functionality without needing to know your password.

In this chapter we're going to look at, on a high level, how OAuth works and then try authenticating with a few social networks using our existing credentials.

Authentication versus authorization

Frequently, there is confusion between the concepts of authentication and authorization. **Authentication** is the act of ensuring that somebody is who they say they are, while **authorization** is the act of ensuring that the person has the rights to perform an action. The concepts are related and have frequently been part of the same step. OAuth breaks the relationship between the two. Although there is typically an authentication step where logging into the server is required, the way that authentication is performed is not prescribed. If the user is already logged into the server site, the authorization step may be transparent to the user. The server can use any method it wishes to perform authentication. This opens the door to allowing for multi-factor authentication or even additional delegation of authentication. For instance, your visualization could request information from Stack Overflow that uses OpenID to delegate authentication. Thus, when a user requests access to Stack Overflow data, the user might actually need to log in with their Google account that would pass authentication details to Stack Overflow, which in turn would pass authorization credentials to your visualization.

It is important to keep in mind the differences between authentication and authorization when acquiring data for your visualization.

The OAuth protocol

Before the introduction and extensive update of OAuth, each service with which you needed to interact provided their own authorization protocol. These methods diverged significantly. Every time you wanted to make use of a new API, a new authorization scheme had to be learned and implemented. This made interacting with a significant number of services very difficult.

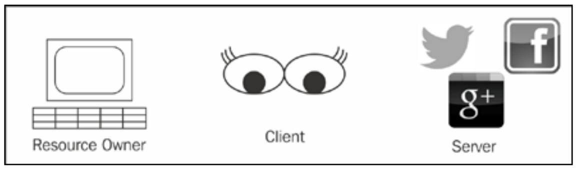
OAuth was created to solve the problem of a lack of standardization around authorization with different sites. The creation of Version 1.0 took about two years and was contributed to by a number of major industry players, as well as smaller interested parties.

OAuth versions

There are currently two major versions of OAuth in the wild: 1.0a and 2.0. Version 1.0a is a security update to the 1.0 specification which corrected a session fixation attack. The 2.0 specification diverged significantly from 1.0a and, unfortunately, there remains a mixture of services that use different protocols in the wild. There is some political debate about the security of the 2.0 specification, which has resulted in a number of companies remaining on the 1.0a version. The biggest difference is that OAuth 1.0 was designed to be usable over an unencrypted connection. When OAuth 1.0 was created, SSL certificates were prohibitively expensive for many small providers. Thus, a complicated series of signed requests were specified which allows even unsecure connections to act as if they were secure.

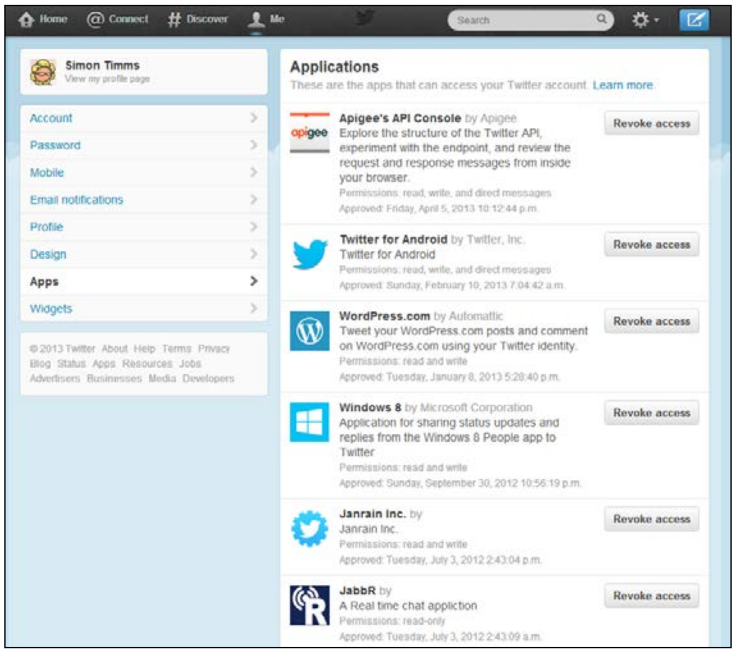
Now SSL certificates have become cheaper, and hardware fast enough, that acquiring an SSL certificate is well within the means of even the smallest startup. Thus, Version 2.0 of OAuth relies on SSL to provide much of its security. It is not safe to use OAuth 2.0 over an unencrypted link. For our purposes, we can remain largely ignorant of the other differences between OAuth versions. However, you should be aware that the authorization and authentication tools you write for one social media site may not work on others. It is even possible that due to implementation differences, your authorization methods will not work between two different sites that both fully support OAuth 2.0.

OAuth defines some different names for the players we saw earlier in the chapter, so let's make use of these terms. An updated view of the previous figure looks like the following:



Resource Owner is the person who "owns" the data. For example, you "own" your updates on Facebook. **Client** is the site requesting access to data and **Server** is the one that has both the data and the credential information for the resource owner. The data that is owned is typically called the **protected resource,** as it is what the OAuth layer is protecting. I've left the icons in place from earlier for clarity, but the client doesn't have to be a visualization, neither does the server have to be a social media site.

You've likely already made use of the OAuth protocol, even if you didn't know it. If you've ever granted an application permission to use your Twitter account, you've used OAuth. The official Twitter applications such as **TweetDeck,** make use of OAuth for authorization. Each application requests access to specific functionality from Twitter. If you're a Twitter user, you can see the applications you've authorized to use your Twitter account from the **Settings** panel. Each one of these applications has been granted permission to interact with Twitter as if it was you, as shown in the following screenshot:



Applications

These are the apps that can access your Twitter account. Learn more.

Apigee's API Console by Apigee
Explore the structure of the Twitter API, experiment with the endpoint, and review the request and response messages from inside your browser.
Permissions: read, write, and direct messages
Approved: Friday, April 5, 2013 10:12:44 p.m.

Twitter for Android by Twitter, Inc.
Twitter for Android
Permissions: read, write, and direct messages
Approved: Sunday, February 10, 2013 7:04:42 a.m.

WordPress.com by Automattic
Tweet your WordPress.com posts and comment on WordPress.com using your Twitter identity.
Permissions: read and write
Approved: Tuesday, January 8, 2013 5:28:40 p.m.

Windows 8 by Microsoft Corporation
Application for sharing status updates and replies from the Windows 8 People app to Twitter
Permissions: read and write
Approved: Sunday, September 30, 2012 10:56:19 p.m.

Janrain Inc. by
Janrain Inc.
Permissions: read and write
Approved: Tuesday, July 3, 2012 2:43:04 p.m.

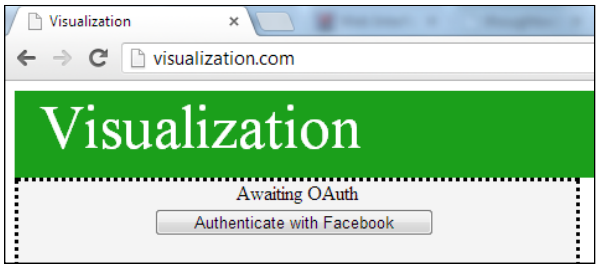
JabbR by
A Real time chat appliction
Permissions: read-only
Approved: Tuesday, July 3, 2012 2:43:09 a.m.

In the preceding screenshot, you'll notice that there is a button next to each application that allows revoking that application's access. This is one of the great features of OAuth—applications never know your password, so you can remove their ability to act as you without having to change your password. If you do wish to change your Twitter password, you only need to do it with Twitter and not with all of the services to which you have granted permission to act on your behalf.

If Twitter were to become aware that, say, `www.wordpress.com` had been compromised, they could revoke the application's access for all their users at once. Keeping credentials is a difficult problem and not one which many developers wish to take on. If credentials can be retained by a reliable company such as Twitter, that removes a common point of failure.

Let's dig a bit more into how OAuth actually works. In order to understand what's happening, it is useful to walk through an example. In this example, our visualization will request some information from Facebook using its Graph API. The **Graph** API is the interface that Facebook have provided developers with to access the social graph, which is really just a collection of properties about a user. Facebook is an OAuth 2.0 site, so this example will use the workflow as described in OAuth 2.0.

Our visualization would like to access information from Facebook. The first time we load the visualization, we're presented with a screen that allows users to click on a button to get access to the Facebook information, as shown in the following screenshot:



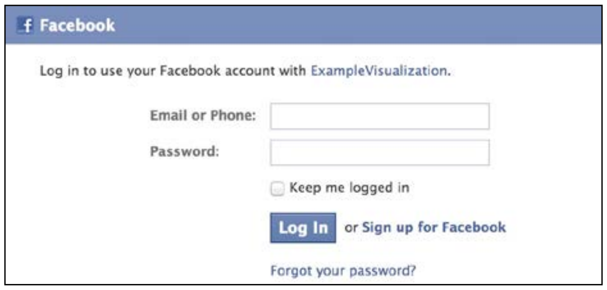
When we first set up our visualization site, we will have requested an authentication token from Facebook. This token is granted by Facebook to our site alone. As part of registering the application with Facebook, we will have entered a domain from which the token can be used. This grants some security over preventing others from using our tokens. Servers may have extensive checks in play to qualify an application to access their protected data.

Our site will generate a request to the Facebook OAuth endpoint which would include the generated token. A typical URL will look like the following:

```
https://www.facebook.com/dialog/oauth?client_id=
591498037422486&redirect_uri=http://visualization.com/&response_
type=token
```

A more detailed example of authenticating against Facebook can be found in *Chapter 7, Facebook*.

The visualization now redirects directly to the Facebook login page, which will ask for your login information, as shown in the following screenshot:



Facebook

Log in to use your Facebook account with ExampleVisualization.

Email or Phone:

Password:

Keep me logged in

Log In or Sign up for Facebook

Forgot your password?

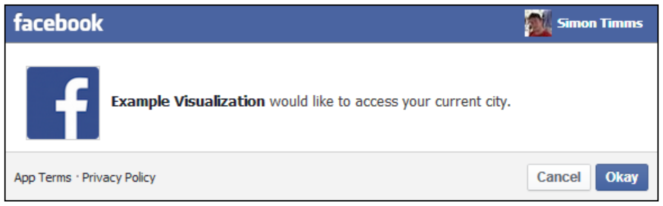
Once you've entered that information and correctly authenticated with Facebook, you'll be redirected back to the page you specified in the `redirect_uri` parameter. This will typically be your original page. Appended to the URI will be a very lengthy token generated by Facebook and used in subsequent communications with their API:

```
http://localhost:8080/Visualization1.html#access_d8Ava9m
MBALMG0FQp22SEn5La7mtC27evICrZB5ToVHdJRZC2FYdFmnIsveVKhcik
SeVZAEAZAHBliKEeGvrKHnb5FnU5VoCooy49FIoJuzca3oHeYuNUZAatdgj
UEr2tDzZBB8CJGmPkHdmNe3RyS1l9XAcTKwGGVAy6FB0gZDZD&expires_in=6667
```

Depending on the permissions your visualization has requested, Facebook may prompt you to grant these permissions explicitly to the visualization. The granting of permissions to applications is they key function of OAuth. Facebook has about 50 different permissions that your application can request. This includes the following:

- `email`
- `user_likes`
- `user_location`

The prompt to grant the application access to your Facebook information looks like the following screenshot:



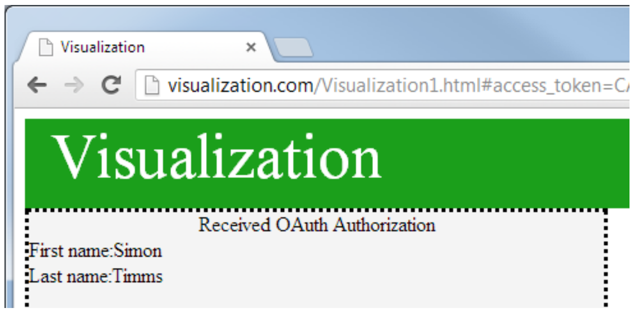
facebook Simon Timms

Example Visualization would like to access your current city.

App Terms · Privacy Policy Cancel Okay

In this example, **Example Visualization** has requested access to my location, which is protected by the `user_location` permission. In Facebook's case, the granted permissions are encoded in the token to be used, but this is not explicitly laid out in the OAuth protocol.

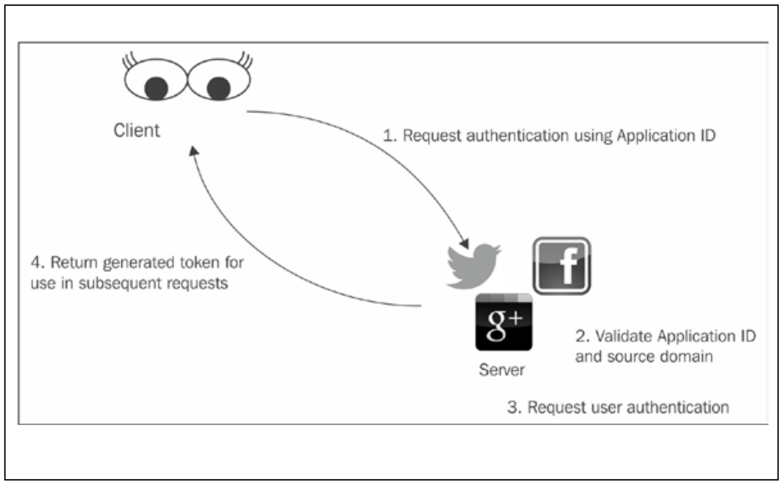
Once permissions are assigned, Facebook will redirect the user back to your `redirect_uri`, allowing you to leverage the token to query the Facebook API, as shown in the following screenshot:



Received OAuth Authorization
First name:Simon
Last name:Timms

For some OAuth sites, it is recommended that an additional call be performed at this stage to ensure that the returned token matches the current application. By sending our application's ID and the returned token from the authentication step, we are provided with some validation details that can be used to confirm the login has not been hijacked. Not every site requires this step; it is simply an added security suggested by Facebook. With the credentials, we can make calls to the Graph API. In this case, the visualization makes a very simple request to retrieve the authenticated user's first and last name. You can see how the request is formulated in the *Chapter 7*, *Facebook*.

That's it! So OAuth delegates the authorization and authentication steps to a third party without the need for complex tools. OAuth uses nothing but normal HTTP and SSL. The workflow for OAuth is almost entirely represented in the following figure:


Client
1. Request authentication using Application ID
2. Validate Application ID and source domain
3. Request user authentication
4. Return generated token for use in subsequent requests
Server

The variety of OAuth dialects and requirements can be difficult to understand. Finding just the right combination of tokens to receive proper authentication with different sites is almost as difficult as the mishmash of techniques used prior to OAuth. OAuth has a reputation for being difficult to understand and inconsistent . This can be attributed to the OAuth standard not being a standard in the same way that HTTP is a standard. With HTTP, if you comply with the standard, you can be assured that your service will be able to interact with all others. OAuth does not offer that same level of interactivity guarantee.

If you're going to be using a lot of different data sources, perhaps if you're creating a mashup, it may be advisable to use a third-party service to communicate with the OAuth servers. Even if you're only using a single data source, you may not wish to complicate your development process with figuring out OAuth. Companies such as DailyCred and OAuth.io provide a service that abstracts away the difficulties of dealing with OAuth. They allow for authentication with numerous different OAuth providers through a consistent API. The hard work of fitting into the various OAuth APIs is handled by them leaving you free to concentrate on building your visualization.

Authenticating against Facebook with OAuth.io is as simple as running the following lines of code:

```
OAuth.initialize('<Your Public key>');
OAuth.redirect('facebook', "callback/url");
```

The example visualization used in this chapter is closer to 70 lines of code.

Of course, these services have a monetary cost to them and also provide an additional failure point. As with all things, care must be taken to ensure that a good solution for OAuth is selected.

Summary

While not every social media site makes use of OAuth, knowledge of how OAuth works and how it can be used to facilitate API usage will most likely improve your experience in developing visualizations. You should now be able to explain how OAuth works. In the next chapter we'll take a look at libraries for visualization.

4
JavaScript for Visualization

In *Chapter 2*, *JavaScript and HTML5 for Visualizations*, we looked at the advantages offered by building our visualizations using scalable vector graphics. It should, however, have been clear that building SVGs by manipulating the underlying XML is a frustrating and time consuming exercise. Although there are countless XML manipulation tools, it would be nice to take advantage of the power of an API which is specifically designed for building SVGs instead of a more general language.

There are a number of JavaScript libraries that have been created for the manipulation of SVGs. svg.js (`http://www.svgjs.com/`) and Raphaël (`http://raphaeljs.com/`), both deserve mentions as being excellent tools for drawing. The demos on the Raphaël website are particularly impressive. d3.js offers functionality designed specifically for visualizations and we'll look at that too.

Raphaël

To draw a simple rectangle making use of **Raphaël** is much more comfortable than building the same rectangle in XML. The library can be included either from the disc or from a CDN such as CloudFlare, as shown in the following code:

```
<html>
  <body>…</body>
  <script
    src="//cdnjs.cloudflare.com/ajax/libs/raphael/2.1.0/raphael-
    min.js"></script>
</html>
```

We can draw any shape in the following manner:

```
function drawRectangle()
{
  var paper = Raphael("visualization", 320, 200);
  paper.rect(50, 20, 50, 150);
}
```

This code finds the element with an ID of `visualization` and appends an SVG to it with a dimension of 320 x 200 pixels. It then inserts a new rectangle at `(50, 20)`, with width `50` and height `150`.

If we wanted to create a simple column graph with Raphaël, it would not be difficult. Let's give it a shot. The first thing we'll need is some data. We can use a JavaScript array for now, but in the real world this information could be retrieved from a web service in either JSON or XML form. In this example, we'll pick some months and the associated value, as shown in the following code:

```
var data = [{month: "May", value: 5},
  {month: "June", value: 4},
…
  {month: "September", value: 8}];
```

Now, let's update the function we used above to draw rectangles to deal with data values. First, we'll change the method signature, as shown in the following code:

```
function drawGraphColumn(paper, item, currentColumn, maximumValue)
{…}
```

This function takes in SVG on which to draw, the current item, and a maximum value that is used to calculate the appropriate height. In the body of the function, we'll start by calculating the bar height, as shown in the following code:

```
var barHeight = (500 * (item.value/maximumValue));
```

We've hard coded the maximum height at 500 px, and each bar is simply a percentage of that height equal to the item's value as a percentage of the maximum. We'll use that value to draw the bar, as shown in the following code:

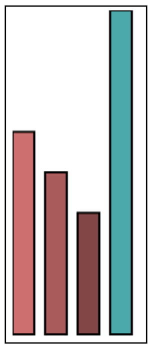
```
var rectangle = paper.rect(currentColumn*30, 500 - barHeight, 20,
  barHeight);
rectangle.attr("fill", "rgb("+ (item.value * 40) + " ," +
  (item.value * 20) + "," + item.value * 20 + ")");
rectangle.attr("stroke-width", 2);
```

The rectangle is offset based on which column it is avoiding overlapping rectangles. We set the color to be a function of the item value so that the color varies by height.

The function is called by passing each one of our data elements into it sequentially:

```
var maximumValue = 0;
$.each(data, function(index, item)
{
  if(item.value > maximumValue) maximumValue = item.value;
});
$.each(data, function(index, item){
  drawGraphColumn(paper, item, index, maximumValue);
});
```

Here, we first calculate the maximum value from the array. Then, call the `drawGraphColumn()` function we defined above for each element. Looping over the data array is done using jQuery's `each` operator which applies the given function to each element in the array. The resulting graph looks like the following:



Raphaël is a general purpose SVG library. This means that apart from being suited to building visualizations, it can be used to create more generic drawings. In the same way that we were looking for an API for manipulating SVGs, which was better suited than manipulating XML by hand, it would be nice to have a library which is targeted at building visualizations. d3.js is a library which is specifically designed to make building visualizations using SVG easier.

d3.js

d3.js brings a number of functions and a coding style that makes creating even simple graphs like the one above simpler. Let's recreate the above graph using d3 and see how it differs. The first thing to do is introduce an SVG element to the page. With Raphaël, we did this using the constructor; in d3, we'll append an SVG element explicitly, as shown in the following code:

```
var graph = d3.select("#visualization")
  .append("svg")
  .attr("width", 500)
  .attr("height", 500);
```

Immediately, you'll see that the style of the JavaScript used differs greatly from Raphaël. d3 relies heavily on the use of method chaining. If you're new to this concept, it is quick to pick up. Each call to a method performs some action, then returns an object, and the next method call operates on this object. So, in this case, the `select` method returns the `div` with the `ID` of `visualization`. Calling `append` on the selected `div` adds an SVG element and then returns it. Finally, the `attr` methods set a property inside the object and then return the object.

At first, method chaining may seem odd, but as we move on you'll see that it is a very powerful technique, and cleans up the code considerably. Without method chaining, we end up with a lot of temporary variables.

Next, we need to find the maximum element in the data array. In the previous example, we used a jQuery `each` loop to find that element. d3 has built in array functions which make this much cleaner, as shown in the following code:

```
var maximumValue = d3.max(data, function(item){ return
  item.value;});
```

There are similar functions for finding minimums and means. None of the functions are anything you couldn't get by using a JavaScript utility library, such as underscore.js or lodash. However, it is convenient to make use of the built-in versions.

The next pieces we make use of are d3's scaling functions, as shown in the following code:

```
var yScale = d3.scale.linear()
  .domain([maximumValue, 0])
  .range([0, 300]);
```

Scaling functions serve to map from one dataset to another. In our case, we're mapping from the values in our data array to the coordinates in our SVG. We use two different scales here: linear and ordinal. The linear scale is used to map a continuous domain to a continuous range. The mapping will be done linearly, so if our domain contained values between `0` and `10`, and our range value ranges between `0` and `100`, then a value of `6` would map to `60`, `3` to `30`, and so forth. It seems trivial, but with more complicated domains and ranges, scales are very helpful. As well as linear scales, there are power and logarithmic scales which may fit your data better.

In our example data, our `y` values are not continuous; they're not even numeric. For this case, we can make use of an ordinal scale, as shown in the following code:

```
var xScale = d3.scale.ordinal()
  .domain(data.map(function(item){return item.month;}))
  .rangeBands([0,500], .1);
```

Ordinal scales map a discrete domain into a continuous range. Here, `domain` is the list of months and the range the width of our SVG. You'll note that instead of using `range` we use `rangeBands`. Range bands split the range up into chunks, each to which each range item is assigned. So, if our domain was `{May, June}` and the range `0` to `100`, then May onwards we would receive a band from `0` to `49`, and June from `50` to `100` would be `june`. You'll also note that `rangeBands` takes an additional parameter; in our case `0.1`. This is a padding value that generates a sort of no man's land between each band. This is ideal for creating a bar or column graph, as we may not want the columns touching. The padding parameter can take a value between `0` and `1` as a decimal representation of how much of the range should be reserved for padding. A value of `0.25` would reserve 25 percent of the range for padding.

There are also a family of built-in scales that deal with providing colors. Selecting colors for your visualization can be challenging, as the colors have to be far enough apart to be discernible. If you're color challenged like me, then the scales `category10`, `category20`, `category20b`, and `category20c` may be for you. You can declare a color scale, as shown in the following code:

```
var colorScale = d3.scale.category10()
  .domain(data.map(function(item){return item.month;}));
```

The previous code will assign a different color to each month out of a set of 10 pre-calculated possible colors.

Finally, we need to actually draw our graph, as shown in the following code:

```
var graphData = graph.selectAll(".bar")
  .data(data);
```

We select all of the `.bar` elements inside the graph using `selectAll`. Hang on! There aren't any elements inside the graph that match the `.bar` selector. Typically, `selectAll` will return a collection of elements matching the selector just as the `$` function does in jQuery. In this case, we're using `selectAll` as a short hand method of creating an empty d3 collection which has a `data` method and can be chained.

We next specify a set of data to union with the data from the existing selection of elements. d3 operates on collection objects without using looping techniques. This allows for a more declarative style of programming, but can be difficult to grasp immediately. In effect, we're creating a union of two sets of data, the currently existing data (found using `selectAll`), and the new data (provided by the `data` function). This method of dealing with data allows for easy updates to the data elements, should further elements be added or removed later.

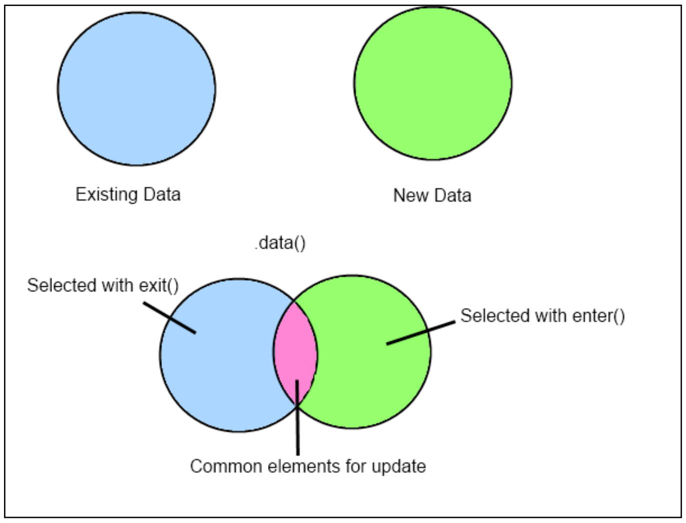
When new data elements are added you can select just those elements by using `enter`(). This prevents repeating actions on existing data. You don't need to redraw the entire image, just the portions that have been updated with new data. Equally, if there are elements in the new dataset which didn't appear in the old one, then they can be selected with `exit()`. Typically, you want to just remove those elements which can be done by the following code:

```
graphData.exit().remove()
```

When we create elements using the newly generated dataset, the data elements will actually be attached to the newly created DOM elements. Creating the elements involves calling `append`, as shown in the following code:

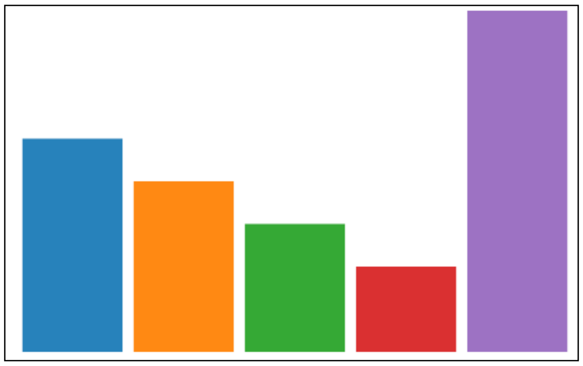
```
graphData.enter()
  .append("rect")
  .attr("x", function(item){ return xScale(item.month);})
  .attr("y", function(item){ return yScale(item.value);})
  .attr("height", function(item){ return 300 -
    yScale(item.value);})
  .attr("width", xScale.rangeBand())
  .attr("fill", function(item, index){return
    colorScale(item.month);});
```

The following diagram shows how `data()` works with new and existing dataset:



You can see in the previous code how useful method chaining has become. It makes the code much shorter and more readable than assigning a series of temporary variables, or passing the results into standalone methods. The scales also come into their own here. The x coordinate is found simply by scaling the month we have using the ordinal scale. Because that scale takes into account the number of elements as well as the padding, nothing more complicated is needed.

The y coordinate is similarly found using previously defined `yScale`. Because the origin in an SVG is in the top left, we have to take the inverse of the scale to calculate the height. Again this is a place where we wouldn't generally be using a constant except for the brevity of our example. The width of the column is found by asking the `xScale` for the width of the bands. Finally, we set the color based on the color scale so it appears like the following graph:



This d3 version of the graph is actually far more capable than the Raphaël version. We eliminated a lot of the magic numbers that were present in the Raphaël through the use of scales.

Let's continue to enhance our graph and explore some of the other features of d3.js.

Custom color scales

The d3 graph we generated above is very colorful, since we made made use of one of the built in color scales. However, most of the time you must have least some thematic consistency in visualizations. You can achieve this consistency across visualizations by leveraging custom scales.

Let's start with a simple example: alternating color scales. In order to make our new color scale, a drop in replacement for the existing `category10` scale we need to use a little bit of JavaScript fun to inject a new scale function into d3. We start with attaching the function to d3's scale namespace, as shown in the following code:

```
d3.scale.alternatingColorScale = function()
{
```

JavaScript allows for monkey patching of objects so this function will actually show up as being part of d3. We'll start implementing the function by setting up the domain and range. We define the `domain` and `range` functions which serve as getters and setters for the domain and range:

```
  var domain, range;
  scale.domain = function(x){
    if(!arguments.length) return domain;
    domain = x;
    return scale;
  }
  scale.range = function(x){
    if(!arguments.length) return range;
    range = x;
    return scale;
  }
```

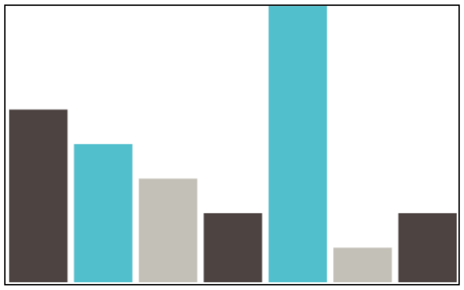
Finally, we'll set up the mapping function which is called when using the scale, as shown in the following code:

```
  function scale(x){
    return range[domain.indexOf(x)%range.length];
  }
  return scale;
}
```

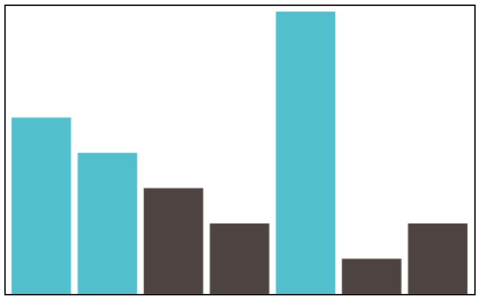
This scale is applied by the following code:

```
var colorScale = d3.scale.alternatingColor()
  .domain(data.map(function(item){return item.month;}))
  .range(["#423A38", "#47B8C8", "#BDB9B1"]);
```

This results in a graph which appears more consistent and is shown in the following graph:



I picked the simplest condition on which to customize our scale, but more complicated and informative scales can be used. A scale which uses a threshold is shown in the following graph:



This is easily done by altering the `scale` function and passing the value instead of the key (month) into the function.

Labels and axes

Up until now, we've build the graphs without much attention to what we're graphing. It would be great to put some labels on the graph, so people can decode our data with ease. Fortunately d3 provides an `axis()` function which makes adding the axis a snap.

We'll start with the x axis, as shown in the following code:

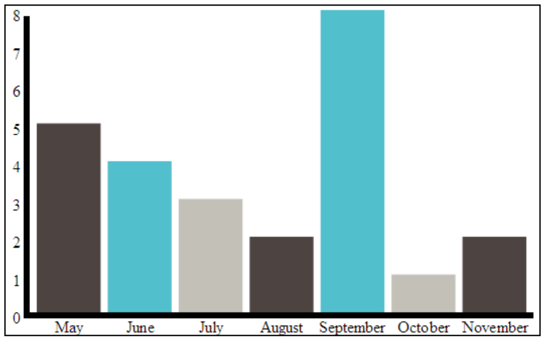
```
var xAxis = d3.svg.axis().scale(xScale).orient("bottom");
  graph.append("g")
  .attr("transform", "translate(20,300)")
  .attr("text-anchor", "middle")
  .call(xAxis);
```

We start here by using the `axis` function to create an axis. We pass in our `xScale` to give it a hint about where the ticks on the scale should be placed. We next append a `g` element to our graph. `g` is an SVG element which acts as a container to hold other elements. You can place any other shape inside the element `g`, and then perform transformations on them as a whole. We do just that in the next step. The axis is, by default, at the top of the graph, so we shift it down and slightly to the right to better line up. The `text-anchor` property sets where the x coordinate of the text should be anchored. By default it is left, but as we have the middle of each bar we set `text-align:middle`. Finally, we pass in the `xAxis`.

The `yAxis` is added just as easily with the following code:

```
var yAxis = d3.svg.axis().scale(yScale).orient("left");
  graph.append("g")
  .attr("transform", "translate(20,6)")
  .call(yAxis);
```

The translation here is a bit more complex, and we're just accounting for the fact that we gave the scale the left orientation, which causes it to be drawn left of the minimum scale value. As our minimum scale value is `0`, it is drawn off screen, as shown in the following graph:



We've managed to add axis and labels in a matter of a few lines of code. With a more general SVG library, this would take a significant amount of work. There are a lot of configuration options for the axis function too. You can set the number of ticks, the labels, and the format of the ticks.

Summary

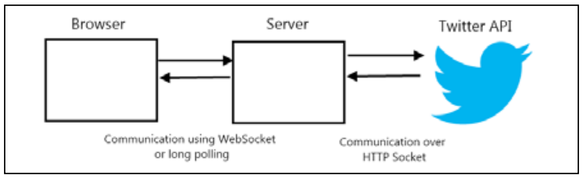
While we've looked at a great deal of d3, we've only just scratched the surface of what d3.js can do. It is a large and powerful library. There are many books written on the subject of d3, so we can't cover it all. A great resource for d3 is Swizec Teller's book *Data Visualization with d3.js, Packt Publishing*. There are tons of additional functions that we'll uncover, as we develop some of the visualizations in the rest of the book. We'll also visit some further applications of the functions presented in this chapter.

5
Twitter

Twitter is a service that really grew up on having an open and available API. Initially, there were no Twitter clients. Communication with Twitter was limited to SMS and later the website. During the development of Twitter, the developers apparently racked up hundreds of dollars in SMS charges, testing, and building the system. Twitter grew in popularity on the back of hundreds of developers who built tools like **TweetDeck** and **Tweetree** using open Twitter APIs or the Twitter RSS feeds. As such, Twitter offers a rich API that can be used to build up applications, and in our case, visualizations.

Before we get into building visualizations, let's take a look at the Twitter API, and how we can make use of it. There is a great deal of documentation on the API available at `https://dev.twitter.com`. If you want more information, this should be your first stop for additional research.

For our purposes, Twitter offers two different models for retrieving data. The first is a typical RESTful model, where the client makes requests to Twitter for specific resources that are returned over HTTP in JSON format. This API is likely to be similar to others you have seen before. It is stateless, meaning that no information is retained server side between requests, and follows the best practices of HTTP. If you're attempting to consume data from Twitter in a web browser, then this is the option for you. The second option is the streaming API. This method makes use of a persistent HTTP connection to which Twitter will send messages as they occur. It is a generally poor idea to make use of the streaming API from a browser, so you'll need to have some sort of an intermediary server between the browser and the API, as shown in the following figure:


Browser
Server
Twitter API
Communication using WebSocket or long polling
Communication over HTTP Socket

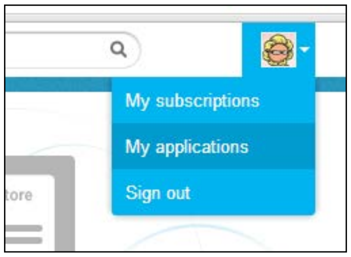
Having a server is, unfortunately, a requirement of all visualizations that make use of Twitter, even the RESTful API, as Twitter does not support authentication using a pure browser-based solution. We'll get into all that later, but first we'll need to set up a developer account with Twitter.

Getting access to the APIs

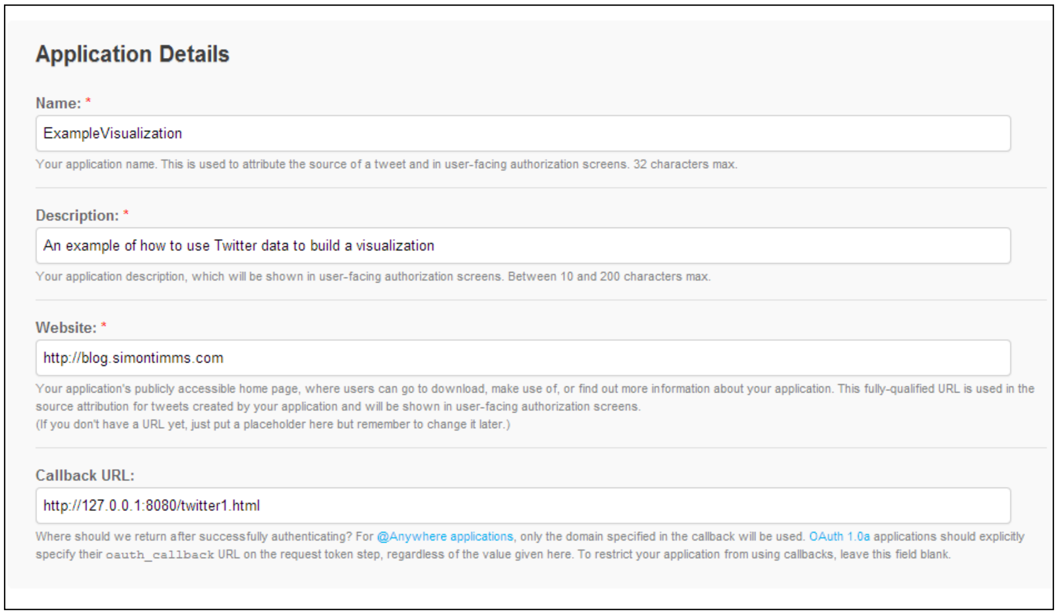
If you remember in *Chapter 3*, *OAuth*, one of the requirements was getting an application key for each site with which we want to talk. This applies to Twitter, so let's go and do that.

Open up a browser, head over to `https://dev.twitter.com`, and click on the **Sign in** link. If you already have a Twitter account, then you can use that here to sign in. If not then you can sign up for a new Twitter account. Don't worry, it is all free.

Once you're signed in, then in the top right-hand corner there should be a link to **My applications,** as shown in the following screenshot:



Clicking on that will take you to a page where you can go about setting up your first application. You'll need to enter some information about the application. You can enter whatever you choose for most fields, but the callback URL should be `http://127.0.0.1:8080/twitter1.html`. This is the URL to which Twitter will direct you once the OAuth phase is complete. We're using a localhost value here, but in production you would want to use the publicly-facing URL for your visualization. The following screenshot shows the **Application Details** window:


Application Details

Name: *

ExampleVisualization

Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.

Description: *

An example of how to use Twitter data to build a visualization

Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.

Website: *

http://blog.simontimms.com

Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL is used in the source attribution for tweets created by your application and will be shown in user-facing authorization screens.
(If you don't have a URL yet, just put a placeholder here but remember to change it later.)

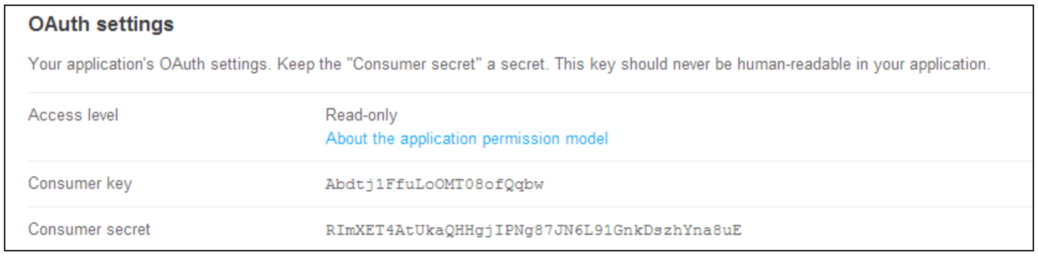
Callback URL:

http://127.0.0.1:8080/twitter1.html

Where should we return after successfully authenticating? For @Anywhere applications, only the domain specified in the callback will be used. OAuth 1.0a applications should explicitly specify their `oauth_callback` URL on the request token step, regardless of the value given here. To restrict your application from using callbacks, leave this field blank.

You cannot use a localhost domain here, but if you would rather not see an IP address, then you can use a URL shortening service to create an alias for your localhost URL. Make sure your URL shortener preserves query parameters, or you won't be able to login correctly.

Once your application has been created, you'll be able to see the various settings from the same **My applications** option that we used previously. The key information for our purpose is `OAuth settings`, as shown in the following figure:


OAuth settings

Your application's OAuth settings. Keep the "Consumer secret" a secret. This key should never be human-readable in your application.

Access level	Read-only About the application permission model
Consumer key	`Abdtj1FfuLoOMT08ofQqbw`
Consumer secret	`RImXET4AtUkaQHHgjIPNg87JN6L91GnkDszhYna8uE`

These keys will be used for user authorization with our application. Should the secret key be leaked—for instance, you might have pasted it into a book you're writing—it can be reset using the **Reset keys** link. Doing so will prevent readers of your book from pretending to be you and committing unspeakable evils in your name.

Setting up a server

As I mentioned, Twitter does not allow direct access to their authentication structure from a browser, we'll need to make use of a server. Fortunately, we can develop against a server on our own computer—no need for an external server. We're using a lot of JavaScript in this book, so let's keep that theme going and host our site locally using node.js. Any other HTTP server will also work.

Installing node.js is pretty simple. If you're on Windows, then there is an installer available from `http://nodejs.org`. On OS X, there is a `.pkg`-based installer available on the same site, or it can be installed using Homebrew. If you're using Linux, it is preferable to compile from source. However, if you're using a distribution with a built-in packaging system, such as **apt** or **yum**, then there is a node.js package that can be installed with either of the two commands:

```
sudo yum install nodejs              #Fedora or RedHat
sudo apt-get install nodejs          #Debian or Ubunt
```

node.js is a piece of server-side software that is designed to perform all of its I/O tasks asynchronously. This means that an operation such as writing to disk is handled without blocking the main thread. When the I/O is complete, the main thread is notified. One of the most common applications is its usage as an HTTP server. This functionally comes in the box in the form of the HTTP module, but the interface provided by that module is pretty lightweight. Instead, we'll make use of the Express framework. Express is a lightweight framework which provides some infrastructure around routing, sessions and serving content, and templating. It can be installed using the node package manager `npm` as shown in the following command:

```
npm install express
```

We'll make use of Express going forward.

OAuth

OAuth can, of course, be manually configured and controlled, but we stand on the shoulders of giants for good reason. It is much easier for us to make use of an already built OAuth library. Fortunately, node has such a library, creatively called **OAuth**. Even with this library, you'll see that interacting with an OAuth 1.0a endpoint is complicated. To install it, drop to the command line again and use the node package manager:

```
npm install oauth
```

This library can perform both OAuth 1.0a and OAuth 2.0 operations. As Twitter is an OAuth 1.0a endpoint, we'll be making use of that.

The first thing to do is set up our Express application. Express provides application templates, but they are an overkill for the simple application in this chapter. If you're planning on creating a more complicated application in future, you'll want to look more into app generation and directory structure for you, application. We start by requiring `express` and creating a new app using the loaded module, as shown in the following code:

```
var express = require("express");
var app = express();
var oAuth = require('oauth');
```

Require is a library which permits the dynamic loading of JavaScript libraries. It is the most common way to bring in external modules in a node application. Next, we configure a number of settings in `express`, as shown in the following code:

```
app.configure(function() {
  app.use(express.bodyParser());
  app.use(express.cookieParser() );
  app.use(express.session({ secret: "a secret key"}));
  app.use(app.router);
  app.use(express.static(__dirname + '/public'));
});
```

`bodyParser` allows `express` for simple parsing of the body of requests sent to the server. On the next line, the `cookieParser` is set up. Like the `bodyParser`, this allows for parsing of cookies and populates the request object with values retrieved from cookies, in our case, information for the session. Next, we set up the session capabilities. This allows us to share information from request to request. In its default configuration, it uses in-memory storage to hold session information. This means that restarting your application will erase session information. If you're hosting your visualization on a farm of machines, you'll need to make use of an external data storage mechanism such as **MongoDB** or **Redis**. We pass in a secret key that is used in generating the `session` hash. It should be a random string. Using the `app.router` will instruct express to listen to route requests, which we'll define in a second. Finally our `.html` and `.js` files are going to be in a directory called `public`, so we'll instruct `express` to serve out the contents of that directory as static resources.

We now want to make use of the OAuth library. This can be done with a function as shown in the following code:

```
function getOAuth(){
  var twitterOauth = new oAuth.OAuth(
  'https://api.twitter.com/oauth/request_token',
  'https://api.twitter.com/oauth/access_token',
  consumerKey,
  consumerSecretKey,
  '1.0A',
  null,
  'HMAC-SHA1');
  return twitterOAuth;
}
```

We create an OAuth object associated with Twitter. We give the two end-points, and then the consumer key and consumer secret which we received from Twitter earlier. The requirement to embed the consumer secret for OAuth 1.0a is why client-side code cannot be used to retrieve information from Twitter. The consumer secret cannot be leaked to outsiders by sending it to the client. `1.0A` is passed in as the version of OAuth; no authorization callback is required so `null` is given as the sixth parameter. The final parameter is the signature method: Twitter uses `HMAC-SHA1`.

Next, we'll set up a route in the Express application to request the OAuth tokens from Twitter:

```
    app.get('/requestOAuth', function(req, res){
      function recieveOAuthRequestTokens(error, oauth_token,
        oauth_token_secret,results) {
        if (!error){
          req.session.oAuthVars = { oauth_token:
            oauth_token,oauth_token_secret: oauth_token_secret};
            res.redirect('https://api.twitter.com/oauth/authorize?oauth_
    token=
            ' + oauth_token);
        }
      requestOAuthRequestTokens(recieveOAuthRequestTokens);
    });
    function requestOAuthRequestTokens(onComplete){
      getOAuth().getOAuthRequestToken(onComplete);
    }
```

Here, we hook up the `/requestOAuth` route to first request an OAuth token, and then use that to redirect the users to the sign in page on Twitter. We build an anonymous function and pass that into OAuth, because node is highly asynchronous. The callback model allows the main node thread to serve another request, while waiting for Twitter to get back to it with OAuth tokens. Once we have the OAuth tokens, we save them in a session state for use in the next step, and redirect to the Twitter authorization page.

Twitter will redirect the user once they have authenticated to the URL we defined when setting up the application. In our case, this will be served by the route `/receiveOAuth`, as shown in the following code:

```
app.get('/receiveOAuth', function(req, res){
  if(!req.session.oAuthVars){
    res.redirect("/requestOAuth");
    return;
  }
  if(!req.session.oAuthVars.oauth_access_token){
    var oa = getOAuth();
    oa.getOAuthAccessToken( req.session.oAuthVars.oauth_token,
      req.session.oAuthVars.oauth_token_secret,
      req.param('oauth_verifier'),
    function(error, oauth_access_token, oauth_access_token_secret,
      tweetRes) {
      req.session.oAuthVars.oauth_access_token =
        oauth_access_token;
      req.session.oAuthVars.oauth_access_token_secret =
        oauth_access_token_secret;
      GetRetweets(res, req.session.oAuthVars.oauth_access_token,
        req.session.oAuthVars.oauth_access_token_secret);
    });
  }
  else
    GetRetweets(res, req.session.oAuthVars.oauth_access_token,
      req.session.oAuthVars.oauth_access_token_secret);
});
```

This code takes the OAuth tokens passed back by Twitter's `redirect`, and performs the final step which is looking up the access tokens. Once we have these access tokens, they can be used to call the API—here done in the `GetRetweets` function. We'll save all the tokens generated in the session so that user's don't have to continually grant access to the Twitter API.

Tired of tokens yet? You should be! This exchange to set up OAuth 1.0a uses an awful lot of tokens. Fortunately, we're done with tokens and OAuth. Now we can get onto building a visualization with Twitter data!

Visualization

There is a whole bunch of APIs made available to us by Twitter. We should start perhaps by inventing something we would like to visualize and then decide if the data is available and how we would show it. I'm curious about which of the people I follow tweet the most. Some accounts such as `@kellabyte` seem to always be tweeting and others like @ericevans hardly at all.

Server side

Let's start by getting the data on the server side. In node.js, I set up a new route using the following code:

```
app.get('/friends', function(req, res){
  if(!req.session.oAuthVars ||
    !req.session.oAuthVars.oauth_access_token){
    res.redirect('/requestOAuth');
    return;
  }
  var cursor = -1;
  receiveUserListPage(res, req.session.twitterVars.user_id,
    req.session.oAuthVars.oauth_access_token,
    req.session.oAuthVars.oauth_access_token_secret, cursor, new
    Array());
});
```

First, we check to make sure that we have the appropriate tokens available in the session. If not then we redirect back to the `requestOAuth` page which will start up the whole OAuth workflow. Next, we set an initial cursor value. Twitter limits the number of results which come back from its services. This avoids dumping a million records out to the consumer, which is not something either party is likely to want. For the API call, we'll be using the limit set to 20. However, Twitter also provides a continuation token which they call a cursor. By calling the service again with this token, the next page of results is returned. An initial value of `-1` gives the first page. The cursor along with all the required tokens is passed into `receiveUserListPage`, which will perform the actual lookups.

Rate Limits

Twitter limits the number of request you can send to their service. When developing a visualization, you may bump into these limits. Wait for 15 minutes and try again. In production, try caching your data so you don't have to query Twitter so frequently.

`receiveUserListPage` looks like the following code:

```
function receiveUserListPage(res, user_id, oauth_access_token,
  oauth_access_token_secret, currentCursor, fullResults){
  var oauth = getOAuth();
  oauth.get(
    'https://api.twitter.com/1.1/friends/list.json?skip_status=true&us
    er_id=' + user_id + "&cursor=" + currentCursor,
  oauth_access_token,
  oauth_access_token_secret,
  function (e, data, oaRes){
  var jsonData = JSON.parse(data);
  if(jsonData.errors){
    projectResults(res, fullResults);
    return;
  }
  fullResults = _.union(fullResults,
  _.map(jsonData.users,
  function(item){return { name: item.name,
    count: item.statuses_count
  }}));
  if(jsonData.next_cursor == 0){
    projectResults(res, fullResults);
  }
  else
    ReceiveUserListPage(res, user_id, oauth_access_token,
      oauth_access_token_secret, jsonData.next_cursor,
      fullResults);
  }
});
}

function projectResults(res, fullResults)
{
  var selectedResults = _.first(_.sortBy(fullResults,
    function(item){return item.count;}).reverse(), 10);
  res.end(JSON.stringify(selectedResults));
}
```

We start off by getting a reference to the OAuth library, then we use the current cursor and the current `user_id` to query Twitter. The API call we're using returns a set of 20 users from the list of people who I follow. The results are returned as a string so we parse them into an object using `JSON.parse`. If the resulting object contains a field called `errors`, then we've likely hit the rate limit, so we return everything we've pull down so far. Because the rate limit is only `15` for this API call, if you follow more than 300 people, you'll hit the limit.

If we have results, we append them to our current set of data. We use the underscore's `map` function to select only two of the fields from. This saves on bandwidth and makes debugging easier, as the objects returned from Twitter are very heavyweight with dozens of useless fields. If `next_cursor` is equal to `0`, then it means that we've reached the end of the list and can return the current set of names and counts. Otherwise we recourse into the function, giving it the new cursor, set of names, and items. Once we hit a case where we can return, we call `projectResults` which sends the 10 users with the most tweets to the client formatted as JSON.

Underscore.js

The underscore JavaScript library is a small library that makes working with arrays far easier. It adds set functions such as `union` and `intersect`, as well as functional programming concepts such as map and reduce. It can be downloaded from `http://underscorejs.org/`.

Client side

The client side visualization code can be placed in the public directory that we previously directed Express to serve out as static content.

I'd like to show the most active tweeters visually. A nice way of doing this is to use a bubble chart, and make the bubbles bigger the more tweets they have. Let's build up the code:

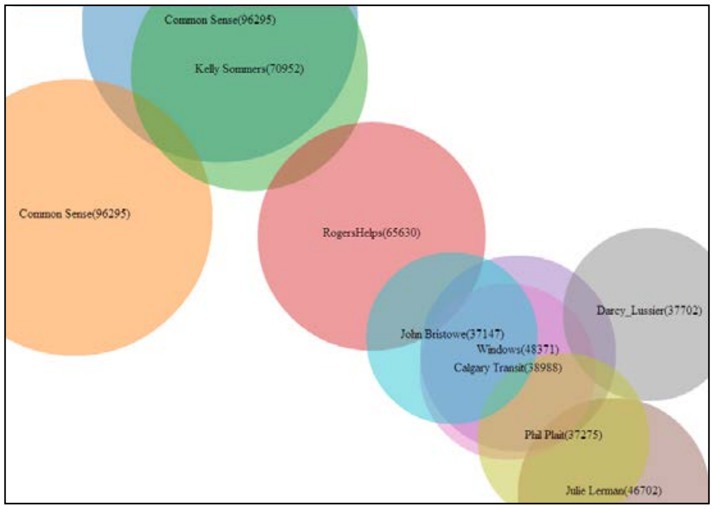
```
function visualize(data){
  var graph = d3.select(".visualization")
  .append("svg")
  .attr("width", 1024)
  .attr("height", 768);
  var colorScale = d3.scale.category10();
  calculateBubbles(data, 1024, 768);
  var currentX = 0;
  graph.selectAll(".bubble")
  data(data)
  enter()
  append("circle")
  .style("fill", function(x,y){return colorScale(y);})
  .attr("cx",  function(d){return d.cx;})
  .attr("cy", function(d){ return d.cy;})
  .attr("r", 0)
  .attr("opacity", .5)
  .transition()
  .duration(750)
```

```
        .attr("r", function(d){return d.radius;});
      graph.selectAll(".label")
        .data(data)
        .enter()
        .append("text")
        .text(function(d){return d.name + "(" + d.count + ")";})
        .attr("x", function(d){return d.cx;})
        .attr("y", function(d){return d.cy;})
        .attr("text-anchor", "middle");
    }
```

Much of this code will look familiar now that you know a bit about d3. The data array which is passed in is what is retrieved from our node service. First, we create an SVG element on the page of an arbitrary size. Then, we set up a color scale so that our visualization will be nicely colored. The `calculateBubbles` function is a helper function which will calculate the locations of the bubbles. It augments our data array with the x and y coordinates for the circle as well as its radius. We won't get into that here, but the code is available on GitHub. For each of the top tweeters we create a bubble. We color it using the color scale, and set the location using the pre-calculated values from the data array. Initially, we set the radius to `0`, but then we use a transition to grow the circles for a nice loading effect.

For each one of the circles, we want to label what the circle represents. This is done by adding a text element at the center of the circle.

The resulting graph, based on the 10 most active people I follow, looks like the following figure:



Every one of these people has tweeted over 35,000 times.

Summary

You should now be able to set up a new application to query Twitter, create the proper OAuth tokens using the OAuth library on node.js, and build a bubble chart. The Twitter API is rich and has many more potential visualizations lurking in it. I'm sure that we could come up with a couple of dozen potential visualizations. There is no better way to learn than through experimenting with the API, so don't be afraid to get messy.

In the next chapter we'll take a look at visualizations of data on the popular question and answer site, Stack Overflow. Their API is largely an open one which doesn't require authentication for most queries, so we should have a brief reprieve from having to use OAuth and even node.js.

6

Stack Overflow

In 2008, the programming question market on the Internet was dominated by a company called Experts Exchange. Many were dissatisfied with the culture on the site and the requirement that people had to be registered to view answers. Programmers Jeff Atwood and Joel Spolsky launched the "question and answer" site, Stack Overflow. Since then, the site has, taken off, quickly growing to become one of the top 100 sites on the Internet. Users can ask and answer questions on the site about a wide variety of programming topics. Answering a question well or asking a well thought-out question wins reputation points, which are prominently displayed. Although, it's not a social media site like Facebook and Twitter, Stack Overflow's content is all user-created and user-moderated. Stack Overflow offers an API against which you can query for all sorts of interesting information.

Authenticating

Much of the query API is available without authenticating. However, if you want private information about users or want to write to the site, then you'll need to authenticate. There is also a much higher request limit for authenticated applications. Without authenticating, a single IP address is limited to 300 requests a day. With an authenticated application, this limit is raised to 10,000 requests.

Rate limits

Many social media sites use rate limits in their APIs. These limits are in place to prevent you from overloading the site, and also to save you from asking for too much data. Twitter processes more than 4,000 tweets a second. Without very special preparation, your infrastructure would quickly be overwhelmed if you were to process them all.

Again, this is a site that makes use of **OAuth** to authorize users. However, they make use of OAuth 2.0, which is much easier than the OAuth 1.0a we used in the previous chapter. We'll limit ourselves to making use of public information to avoid authenticating. Should you wish to authenticate, I promise it is easier than Twitter. You can find instructions at `https://api.stackexchange.com/docs/authentication`. Stack Overflow uses the same authorization system as Facebook, so the example from the OAuth chapter should work perfectly.

Creating a visualization

Many of the questions on Stack Overflow have a large number of answers. The site is not optimized to show the latest answers; the answers are ranked by being the most accepted answer then randomly. This is done to give all answers a chance at being shown near the top which should, in theory, encourage people to vote for the best answer instead of just the first answer shown.

For this visualization, I would like to show how a question has been answered over time. Are more recent answers likely to get a higher score? Is the first answer always the best?

Let's start by pulling down the data for an individual question which has a large number of answers. To do this, we'll make use of the questions API. All of the API endpoints are hosted on `https://api.stackexchange.com`. We're going to make use of the latest API which is Version 2.1. This is also encoded into the URI, as is the specific endpoint and the ID. Within the question API, we're interested in the answers, so we can query specifically for them, giving us a URI of `https://api.stackexchange.com/2.1/questions/{id}/answers`.

In the query string, we'll specify the site against which we want to query. Stack Exchange hosts several dozen question and answer sites modeled on Stack Overflow, all of which are served from the same API endpoint, so it is necessary to filter just for Stack Overflow by passing in `site=stackoverflow`:

```
function retrieveQuestionAnswers(id){
  var page = 1;
  var has_more = true;
  var results = [];
  while(has_more) {
    $.ajax(https://api.stackexchange.com/2.1/questions/ + id +
      "/answers?site=stackoverflow&page=" + page,{
        success: function(json){
          has_more = json.has_more;
```

```
            results = results.concat(json.items);},
          failure: function() {
            has_more = false;},
            async: false
          });
          page++;
      }
    return results;
  }
```

Twitter provided us with the continuation tokens that we could pass back to Twitter to request the next page of data. Stack Overflow takes a different approach and assigns page numbers, allowing us to browse through the results with ease. Embedded in the response for every API call is a token called `has_more`, which is true whenever there are more pages of data that match the current query.

In this code, we make use of the continuation token and the page number to perform as many queries as necessary to retrieve all the answers. We are making use of the jQuery function `ajax,` instead of the more common `getJson` function, because we would like to retrieve the data synchronously. We do this because we want the entire dataset at one time. If your visualization allows for data to be added dynamically then you can relax the `async:false` requirement.

What's returned is an array of objects, each one of which represents an answer to a question. If we give the `retrieveQuestionAnswers` method an ID such as `901115`, then we'll get back an array of 50 answers. These come back over the course of two requests and the code above merges them together into the results array which is returned.

Each `Answer` contains a number of fields. A list of the fields returned by default can be found at `https://api.stackexchange.com/docs/types/answer`. For the purpose of our visualization, we're most interested in when the answer was originally suggested, its score, and also whether it was chosen as the accepted answer. These bits of information can be found in the fields: `creation_date`, `score`, and `is_accepted`. We'll ignore the rest of the fields for now.

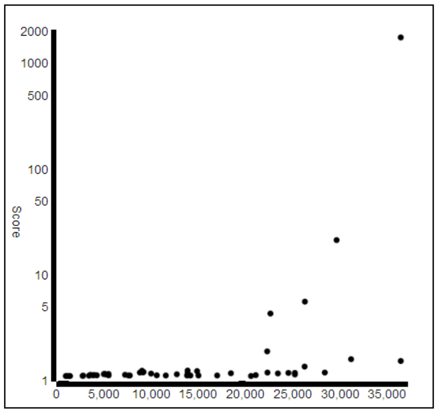
Now that we have some basic data, we can start thinking about the visualization. We're trying to convey the relationship between the age of a question and its score. This sounds a lot like a use for a scatter plot. The data points stand on their own and can be placed along two axis, date and points. My theory before starting on this that answers that are older will tend to have a higher score, because they've been around longer to gather points. People are programmed to believe that numbers going up are positive, so let's play to that and plot points versus age which will, if my theory holds, have higher values on the right.

Of course, a scatter plot is boring and nothing we couldn't generate outside of Excel. We'll add some interactivity to it, but to start, we'll still need a simple scatter plot.

This is easily done with a couple of scales and some circles, as shown in the following code:

```
var graph = d3.select("#graph");
var axisWidth = 50;
var graphWidth = graph.attr("width");
var graphHeight = graph.attr("height");
var xScale = d3.scale.linear()
  .domain([0, d3.max(data, function(item){ return item.age;})])
  .range([axisWidth,graphWidth-axisWidth]);
var yScale = d3.scale.log()
  .domain([d3.max(data, function(item){return item.score;}),1])
  .range([axisWidth,graphHeight-axisWidth]);
```

This gives a very flat graph with the majority of the data being close to zero, while the scale is skewed by a high outlier with a score over 2000, as can be seen in the following figure:



This can be ameliorated by using a logarithmic scale. Any time you use a non-standard scale like logarithmic, you'll want to put in axis labels to prevent causing confusion or misleading the consumer of the visualization.

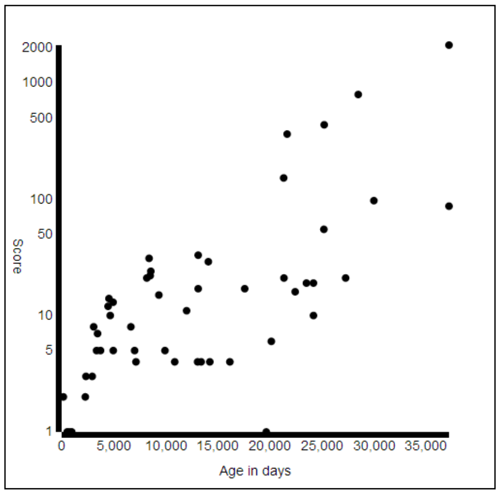
```
var yAxis = d3.svg.axis()
  .scale(yScale)
  .orient('left')
  .tickValues([1,5,10,50,100,500,1000,2000])
  .tickFormat(function(item){return item;});
```

```
graph.append("g")
  .attr("transform", "translate(" + axisWidth +",0)")
  .call(yAxis);
graph.append("text")
  .attr("x", "0")
  .attr("y", graphHeight/2)
  .attr("transform", "rotate(90, 0, " + graphHeight/2 + ")")
  .text("Score");
```

The labels in this graph are manually assigned to give the best spread. You can automatically assign labels, but I found them to be declared at odd places. I also defined a function to format the labels, otherwise they had a tendency to be formatted using scientific notation (*2 * 10^3*). Finally, I appended some text as an axis label. I also added an age axis that lists the age of the answer in days.

```
var xAxis = d3.svg.axis().scale(xScale).orient('bottom');
graph.append("g")
  .attr("transform", "translate(0," + (graph.attr("height") -
    axisWidth)  +")")
  .call(xAxis);
graph.append("text")
  .attr("x", graphWidth/2)
  .attr("y", graphHeight-5)
  .style("text-anchor", "middle")
  .text("Age in days");
```

The only special thing worth noting in this code is that the label is rotated using a transform, as it appears along a vertical axis. The resulting graph looks like this figure:



Now that we have a basic visualization, we can start spiffing it up with some interaction.

The simplest interaction we can add is to pop up a label when somebody moves the mouse pointer over one of the points.

This can be done by using the `on()` function of `d3`. This function can tie event listeners to the elements created as part of an SVG. To start, we add to the end of circle appending from above, as shown in the following code:

```
//append circle
.on("mouseover", function(item){
  showTip(item);
});
```

Here, the `showTip()` function will be called whenever the user hovers the mouse over one of the circles in the above graph. The `item` parameter, which is passed into the event handler, is the item from the data collection that is attached to the hovered circle. If you need additional information about the event, and we do, then that can be found attached to the global variable `d3.event`.

In the event handler, we first highlight the selected circle by ensuring all other circles are black and then making the selected one blue:

```
function showTip(item){
  d3.selectAll(".score").attr("fill", "black");
  d3.select(d3.event.srcElement).attr("fill", "blue");
```

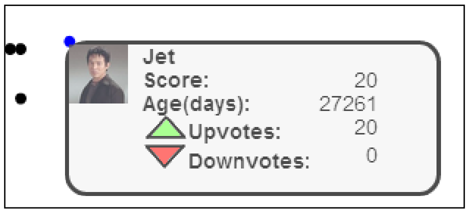
It may also be useful to change the size of the circle to draw even more attention to it. This can be done by simply updating its attributes. Next, we hide the previous tip and set the inner contents of the tip to take values from the selected data element:

```
  d3.select("#tip").style("opacity", 0);
  d3.select("#count").text(item.score);
  d3.select("#age").text(Math.floor(item.age));
  d3.select("#profileImage").attr("src",
  item.owner.profile_image);
  d3.select("#profileName").text(item.owner.display_name);
```

Finally, we move the tool tip to be next to the circle and have it fade in:

```
  d3.select("#tip").style("left", d3.event.x + "px");
  d3.select("#tip").style("top", d3.event.y + "px");
  d3.select("#tip").transition().duration(400).style("opacity",
    .75);
}
```

The end result looks like the following diagram:



Adding interactivity to your visualization allows you to present far more data than would normally be possible. Hiding data so that it can only be seen by moving the mouse over, or clicking on it prevents overwhelming your users while still providing the maximum amount of information.

Filters

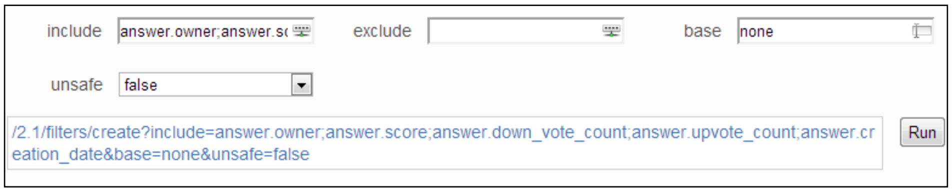
The data returned by our query isn't exactly what we want. For instance, we don't care about `last_edit_date` or even the `last_activity_date`, but we do care about the number of up and down votes. By pulling extra data back, we're wasting bandwidth and slowing down the visualization for our users. Fortunately, Stack Overflow has a solution for that in the form of filters.

Deep queries

If you find that you need to explore the Stack Overflow data in greater depth than is provided for by the API, you can download a dump of the entire site at `http://www.clearbits.net/creators/146-stack-exchange-data-dump`. This dump is provided every three months, and currently clocks in at 13.4 GB compressed. With this dump, you can run much more complex queries without the fear of hitting a rate limit.

Filters govern what data is returned from the API, and can be used to either add or remove fields. They are statically created so you should only need to create them once and there is no need to create a new filter each time you query the site, or even each time your application is launched. In fact, I actually make use of the API explorer provided by Stack Exchange to create my filters ahead of time. The URL for creating filters is `https://api.stackexchange.com/docs/create-filter`.

In the **include** field, you can place a semi-colon to include a delimited series of names. Everything which is part of the answer object is prefaced by answer, so the answer owner would be referred to as `answer.owner`. The default filter is quite inclusive so as a base filter I've used the special `none` filter. This includes no fields unless they are explicitly included. Using the `none` filter as a base is the best practice to reduce excess queries, as shown in the following figure:



include answer.owner;answer.sc
exclude
base none
unsafe false

```
/2.1/filters/create?include=answer.owner;answer.score;answer.down_vote_count;answer.upvote_count;answer.cr
eation_date&base=none&unsafe=false
```

If you do start with the `none` filter, be sure to add the tokens `.items` and `.has_more` to the include list. Without items, the items collection—which holds either questions, answers, or users depending on the query—isn't included and `has_more` is needed to tell if there are additional pages. For our purposes, the following filter is perfect:

```
answer.answer_id;answer.owner;answer.score;answer.down_vote_
count;answer.upvote_count;answer.creation_date;shallow_user.profile_
image;shallow_user.display_name;.items;.has_more
```

The `create` filter returns an alpha-numeric string which can then be used in our query to filter it appropriately. The URL against which we're querying becomes the following:

```
"https://api.stackexchange.com/2.1/questions/" + id + "/answers?site=s
tackoverflow&filter=!2BjddbKa0El(rE-eV_QT8)5M&page=" + page
```

By using a filter, I was able to reduce the payload returned from the API to 3kB from 22kB. This is a significant saving, especially over low bandwidth connections.

Summary

You should now be able to query against the Stack Exchange API for not just the Stack Overflow but for all the Stack Exchange sites. You should also have some idea of how to add interactivity to your visualizations through the use of `d3`. In the next chapter we'll take a look at using Facebook as a source of data to visualize.

7 Facebook

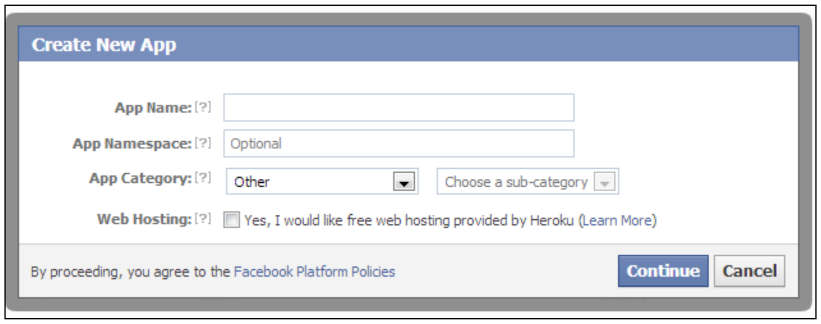
Facebook is the 900 lbs (408.233 kg) gorilla of the social media world. Literally created in a university dorm, Facebook has grown to have 1.1 billion active users. That's one out of every seven people on the planet. It has impressive sway to say the least. No book about using social media APIs could be complete without an investigation of how to use Facebook's API.

Creating an app

As you might expect for such a large site, there are numerous APIs available for building applications related to Facebook. The simplest involve integrating "post to Facebook" buttons in websites or mobile apps, and the most complicated allow you to actually run the code on Facebook's servers as an app. We're going to make use of the Graph API.

The Graph API provides an HTTP-based method of accessing information, what Facebook calls the social graph. The graph is really just the relationship between various users and their data. It is a graph and a collection of nodes and edges as opposed to the bar graph-style of chart.

To get started, we'll register an application with Facebook, much as we did with Twitter and would have to do with Stack Overflow if we wanted to make use of authentication. To do this we'll head over to `http://developers.facebook.com` and click on the **Apps** link in the top menu bar. From there, click on **Create New App**. You'll be presented with the **Create New App** dialog box as shown in the following screenshot:



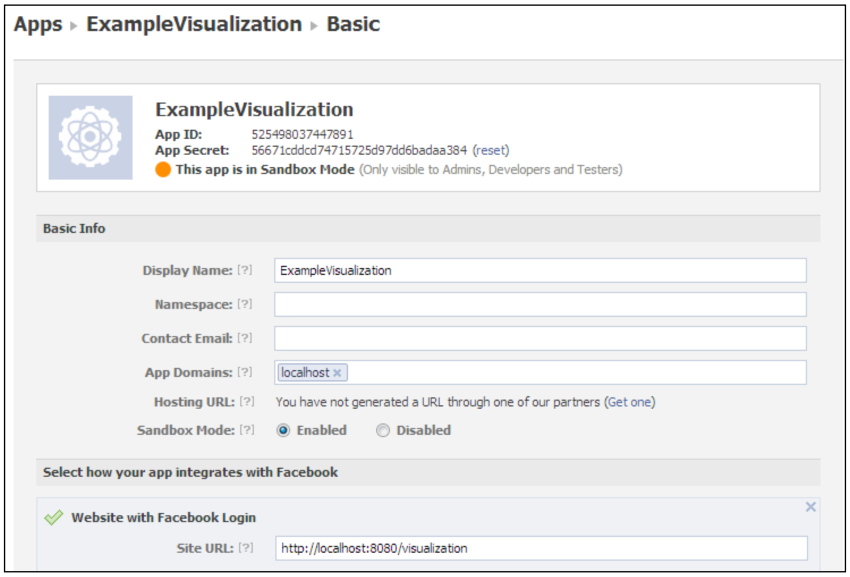
Create New App

App Name: [?]	
App Namespace: [?]	Optional
App Category: [?]	Other / Choose a sub-category
Web Hosting: [?]	Yes, I would like free web hosting provided by Heroku (Learn More)

By proceeding, you agree to the Facebook Platform Policies

The app name can be anything you like, the app namespace is used to give your app a location on Facebook such as `apps.facebook.com/NiftyVisualization`. It is largely unnecessary for our purposes. The app category is wholly up to you and should be determined based on what it is that you're visualizing. **Heroku** is a cloud-based hosting provider partnered with Facebook to provide hosting space for Facebook applications. If you don't have hosting already, Heroku is a reasonable alternative and does support node.js; however, using it is outside the scope of this book.

Once you've filled out the details of your application, you'll be asked to confirm that you're a human by solving a CAPTCHA puzzle. You'll now be taken to the edit page where you can fill in the last few details before testing out your access to the API. It looks like the following screenshot:


Apps ▸ ExampleVisualization ▸ Basic

ExampleVisualization
App ID: 525498037447891
App Secret: 56671cddcd74715725d97dd6badaa384 (reset)
This app is in Sandbox Mode (Only visible to Admins, Developers and Testers)

Basic Info

Display Name: [?]	ExampleVisualization
Namespace: [?]	
Contact Email: [?]	
App Domains: [?]	localhost
Hosting URL: [?]	You have not generated a URL through one of our partners (Get one)
Sandbox Mode: [?]	[✓ Enabled] Disabled

Select how your app integrates with Facebook

✓ Website with Facebook Login
Site URL: [?] http://localhost:8080/visualization

Here, you'll need to fill in at least one app domain. This value is checked by the API when you sign in to ensure that your app is being used from an authorized domain. Unfortunately, you can't access Facebook's Graph API from a file that isn't served from a domain. This means that just going to `file://c:/code/visualization.html` won't allow you to access Facebook's API. Fortunately not all is lost, using `localhost` is permitted, but that does mean we have to run an HTTP server. We can make use of the same `Node.js` installation which we used in previous chapters.

The site URL in the website with the Facebook login should be set to be the return URL for your OAuth key exchange. We can actually set this to anything because we're going to use AJAX to do the authentication, and our users won't ever actually move away from our initial page.

Using the API

It is perfectly possible to manually perform OAuth to authenticate and authorize your visualization with Facebook. However, Facebook has been kind enough to provide a very usable JavaScript SDK. The API abstracts away the process of logging into a function call. To make use of the API, we first need to include it in our visualization. To do this, simply include the following script inside one of your `script` tags:

```
(function(d){
  var js;
  var id = 'facebook-jssdk';
  var ref = d.getElementsByTagName('script')[0];
  if (d.getElementById(id)) {return;}
  js = d.createElement('script'); js.id = id; js.async = true;
  js.src = "//connect.facebook.net/en_US/all.js";
  ref.parentNode.insertBefore(js, ref);
}(document));
```

This code will create a new `script` tag in your document and set its source to a file on the Facebook site. Adding a `script` tag like this will cause the browser to load the contents of that script file and execute it. Because we're loading the script in an asynchronous fashion, we'll need to wait for it to be loaded before we make use of it. The SDK calls a hook, `fbAsyncInit`, once it is initialized. We just need to tie a function to that hook, as shown in the following code:

```
window.fbAsyncInit = function() {
  FB.init({
  appId      : '525498574499442',
  channelUrl : '//localhost:8080/channel',
  status     : true,
  cookie     : true,
  xfbml      : true  // parse XFBML
  });
};
```

This will provide the SDK with the App ID available from the developer's webpage. Also, here I've provided `channelUrl` that is used to solve some cross-domain issues that appear on some browsers. Setting the status will have the `init` method fetch the value of `status`. `cookie` will enable cookie support. Finally `xfbml` enables Facebook markup language. What's that? It is a collection of specially formatted HTML elements that are controlled by the Facebook SDK.

For instance, if we would like to show a login button (how convenient that we do want to show a login button) then we can simply add the following code:

```
<fb:login-button show-faces="true" width="200" max-rows="1"
  scope="user_birthday,email,friends_birthday">
</fb:login-button>
```

When an unauthenticated user opens the page, then a login button will show. When an authenticated user reaches the page, they will be shown their own login information. You'll note the `scope` attribute; this is used to give Facebook a list of permissions you're requesting. Here we've asked for the logged in user's birthday, e-mail ID, and their friends' birthdays. When signing in, the user will be prompted by Facebook to allow your visualization access to that permission. There are about three dozen different permissions which can be requested from Facebook that govern everything from retrieving information about the logged in user, to their friends, events, and RSVPs. It is well worth poking around in here to discover interesting aspects to visualize.

The final piece in the authentication puzzle is to provide a function for the login button to call once it has logged the user in:

```
FB.Event.subscribe('auth.authResponseChange', function(response) {
  if (response.status === 'connected') {
    //use SDK here
  } else if (response.status === 'not_authorized') {//not
    authorized
    FB.login({scope: "user_birthday,email,friends_birthday"});
  } else { //not logged in
    FB.login({scope: "user_birthday,email,friends_birthday"});
  }
});
```

This event is triggered whenever there is a change in the authorization response, such as when we get authentication back from the login button.

Retrieving data

Before we go about retrieving data we should probably decide what data we wish to visualize. The amount of data which is available about the logged in user is not all that great (at least it isn't for me but I hardly use Facebook). That leaves us with looking at our friends. I found devices my friends use to access Facebook to be quite interesting. Are they more Android users or iOS? This information is available as part of the friends' collection. To retrieve this information we can use the `FB.api()` method:

```
FB.api('/me?fields=friends.fields(devices)', function(response){
  for(i = 0; len = response.friends.data.length; i< len; i++){
    var friend = response.friends.data[i];
    if(friend.devices)
      for(j = 0; j< friend.devices.length; j++)
    if(friend.devices[j].hardware != "iPad")
      operatingSystems[friend.devices[j].os]++;
  }
});
```

Into the `api()` method we pass a URL to be retrieved. In this case, we request the special URL `/me` which refers to the currently logged in users. We also provide a filter so that only the friends' collection is retrieved and, in fact, only the device's collections are retrieved for each friend. In the callback, we're just counting up the number of Android versus iOS devices. iPads and iPhones are separate devices to Facebook, but we don't want to count iOS as an access method twice, so we ignore any iPads. Once this code executes, we end up with a collection of device counts. For my friends I got the following:

```
{Android: 28, iOS: 36}
```

Visualizing

One of the more effective visualization techniques is to show the relative strengths of different categories by showing a scaled image. We saw this technique applied in the Twitter chapter using bubbles. We can take that to the next step by using images instead of just circles.

The first step is to locate logos for Android and iOS that are already SVGs. As it turns out, Wikipedia is a great source for this and their images are all licensed under creative commons, meaning we can use them in our visualizations. One of the really great features of SVG is that you can easily merge two images together through the use of definitions. If you open up an SVG like the Android logo at `http://upload.wikimedia.org/wikipedia/commons/e/e1/Android_dance.svg`, you can copy all the markup under a `<defs>` tag in another image. I took the Android and Apple logos and moved them into my raw markup. If I wanted to display them, I could use the `<use>` tag and reference the definitions by ID. It looks like the following code:

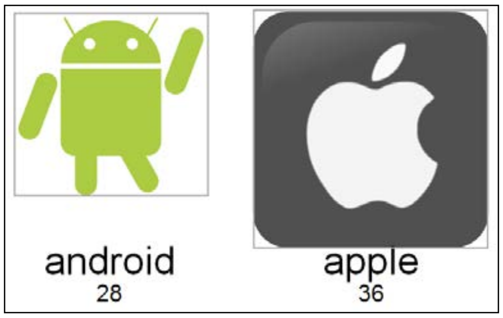
```
<defs>
  <g id="appleLogo">
    <!--various shapes needed to build the Apple logo-->
  </g>
  <g id="androidLogo">
    <!--various shapes needed to build the Android logo-->
  </g>

</defs>
<use x="0" y="10" xlink:href="#appleLogo"/>
<use x="512" y="10" xlink:href="#androidLogo" />
```

This will create an Apple logo next to an Android logo in our SVG. Knowing that we can leverage `d3` to build and scale the logos as appropriate, we're lucky, in that both of the SVGs we have are 256 px square, so they look to be approximately the same size before we've translated them. The `d3` is relatively simple, as shown in the following code:

```
var visualization = d3.select("#visualization");
visualization.selectAll(".logo").data(operatingSystems)
.enter().append("use")
.attr("xlink:href", function(item){ return "#" + item.os +
  "Logo";})
.attr("transform", function(item, index){
  return "translate("  + 300 * index + " 0),scale(" + (item.users
    / operatingSystems[0].users) + ")";
});
```

We start by selecting the SVG then instead of appending shapes, we append using statements. The `xlink:href` attribute takes the value of the definition to include. Next, we scale and translate the logos so they are next to each other and the appropriate size. We set the first logo to be the baseline size, and every subsequent logo is drawn as a percentage of that. This only works because our numbers are quite close. With highly divergent numbers, a more robust strategy would be needed. With some additional text elements appended, the result is the following figure:



Summary

You should now have a grasp of how to make use of the Facebook API to retrieve data. This data can then be visualized using any technique. In the next chapter we'll take a look at the upstart Google+ social network and see how we can leverage the data present there for visualizations.

8
Google+

Of the major social networking sites, Google+ is the newest entrant. Although an upstart, it does have a large user base claiming to have more than 350 million active accounts (`http://ca.ign.com/articles/2013/05/02/report-google-bigger-than-twitter-with-359-million-active-users`). This is not Google's first attempt at breaking into the billion-dollar social media market. They have, in the past, created Google Buzz, Google Friend Connect, and Orkut in an attempt to gain a large user base. All but Orkut have since been mothballed and its user base is almost entirely located in Brazil. Google has purposefully avoided creating a write API in the hope that it eliminates automatically-posted spam. Google+ provides a read-only API that we can leverage to create visualizations; however, the API is very limited in comparison to other such APIs—while this book is being written you cannot even list the members of a circle.

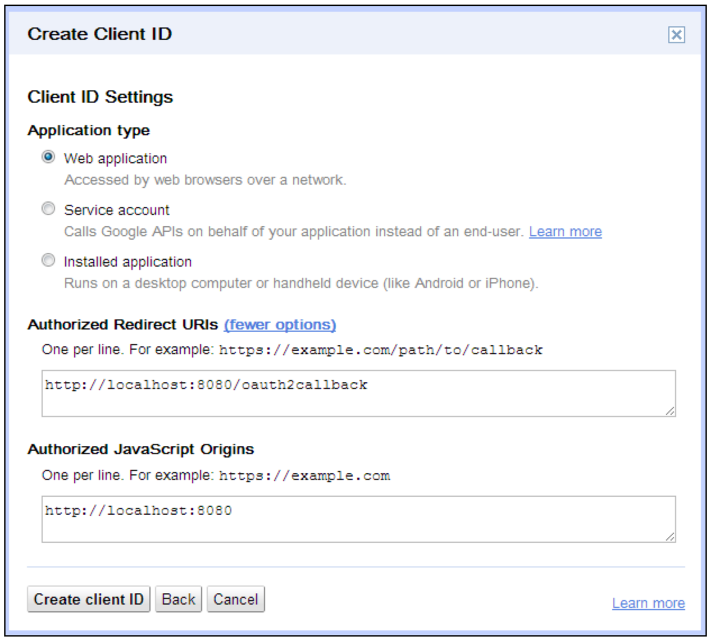
Creating an app

Google+ is another OAuth 2.0 site so we, of course, need to get an application key as the first step to creating any visualization. This also means that we will need a return URL, so again we'll need to set up an HTTP server to run the visualization.

The first step is to log into `https://code.google.com/apis/console` using your Google account. Should you not have such an account, you can also create one from that page. Once on the site, you'll be presented with a giant button allowing you to create an application project. This console actually governs the access to all of Google's APIs, and there are quite a few.

Next, you'll be presented with a huge list of the various APIs. If you scroll way down, you'll eventually find Google+ (use the search, it will save hours of scrolling). Toggle the switch to the "on" position. You may need to agree to a couple of user agreements. Be sure to read the entire agreement as you always do.

The next step is to request a new key, as shown in the following screenshot. This can be done from the **API Access** tab in which you should click on **Create an OAuth 2.0 client ID...**. In the dialog that opens, you'll need to fill in an application name and a URL. This is not the URL for the OAuth exchange; that comes in the next tab. On this tab, enter a URL from which the OAuth request may originate and to which it should return. For our purposes, `http://localhost:8080` will be the domain:


Create Client ID

Client ID Settings

Application type

(●) Web application
Accessed by web browsers over a network.

( ) Service account
Calls Google APIs on behalf of your application instead of an end-user. Learn more

( ) Installed application
Runs on a desktop computer or handheld device (like Android or iPhone).

Authorized Redirect URIs (fewer options)
One per line. For example: https://example.com/path/to/callback

http://localhost:8080/oauth2callback

Authorized JavaScript Origins
One per line. For example: https://example.com

http://localhost:8080

We'll then receive a couple of keys ready for use in our application. The client ID is the field you want to use in your scripts.

Retrieving data

As with Facebook, we could do manual authentication against the OAuth 2.0 endpoint, but let's make use of Google's provided API. Hooking it up is very simple:

```
document.addEventListener("DOMContentLoaded",
 function() {
    var po = document.createElement('script');
    po.type = 'text/javascript'; po.async = true;
    po.src = 'https://plus.google.com/js/client:plusone.js';
    var s = document.getElementsByTagName('script')[0];
    s.parentNode.insertBefore(po, s);
  });
```

This runs once the document is ready and loads the API script from Google's servers by placing a new `script` tag on the page just before the tag including jQuery. The loaded document contains a number of JavaScript functions that can be used to interact with Google APIs but not specifically Google+ APIs—that happens after login.

To add a login button, we add the following HTML code:

```
<button class="g-signin"
   data-scope="https://www.googleapis.com/auth/plus.login"
   data-requestvisibleactions="http://schemas.google.com/AddActivity"
   data-clientId="988552574579.apps.googleusercontent.com"
   data-callback="onSignInCallback"
   data-theme="dark"
   data-cookiepolicy="single_host_origin">
</button>
```

This will generate the login button. The various `data-` properties attached to this button are processed by the script that we loaded from Google. The login is scoped to Google+ instead of one of the other Google APIs. The Client ID should be set to the one that was retrieved when creating the app. Most importantly, a callback function is assigned that will be activated when the request to log in succeeds. The callback will dynamically load the Google+ API:

```
function onSignInCallback(authResult) {
      gapi.client.load('plus','v1', function(){
        if (authResult['access_token']) {
          $('#gConnect').hide();
                           retrieveFriends();
        } else if (authResult['error']) {
          console.log('There was an error: ' + authResult['error']);
          $('#gConnect').show();
        }
        console.log('authResult', authResult);
      });
    }
```

Once the API for Google+ is loaded, we can make use of it, as we are doing in the highlighted line. This function also hides the sign in button, so users don't attempt to sign in more than once. `retreiveFriends` is simple and will just send off a request to retrieve a list of friends:

```
retrieveFriends: function(){
      var request = gapi.client.plus.people.list({
        'userId': 'me',
        'collection': 'visible'
      });
      request.execute(retrieveFriendsCallback);
    }
```

Now that we have a list of friends, we can set about building a simple visualization using them.

Visualization

`d3` purposefully steers away from providing concrete visualizations. There is no single function you can call to get a bar chart or a scatter plot. Instead, it provides tools around creating the visualizations; these tools in turn provide a high degree of flexibility and empowers the creation of unique visualizations. One of the more powerful tools is the layout mechanism. Layouts provide some of the boilerplate code that would have to be written to achieve a certain sort of visualization.

We're going to make use of the **Force-directed graph** layout. Force-directed graphs provide a way of visualizing data that is interconnected. The strength of the bonds between nodes is frequently a function of how closely related the nodes are.

Our first step is to transform our data into a list of nodes and edges. As the API returns such limited data, we'll only be able to establish relationships between you and your friends. Those relationships will make up the edges or links, and the friends, the nodes:

```
var nodes = [];
 var links = [];
var centerNode = { name: "Me"};
nodes.push(centerNode);
for(i = 0; i< data.items.length; i++){
   var node = { name: data.items[i].displayName, image: data.items[i].
image.url};
   nodes.push(node);
   links.push({source: centerNode, target: node});
}
```

Now that we have the nodes and links, we can create a force layout using them:

```
var graph = d3.select("#graph");
var force = d3.layout.force().charge(-120).linkDistance(100).
size([500,500]).nodes(nodes).start();
```

The `charge` and `linkDistance` functions govern how widely the nodes disperse themselves. For the links, we draw a simple line to represent them:

```
var link = graph.selectAll(".link").data(links).enter().
append("line").attr("class", "link");
```

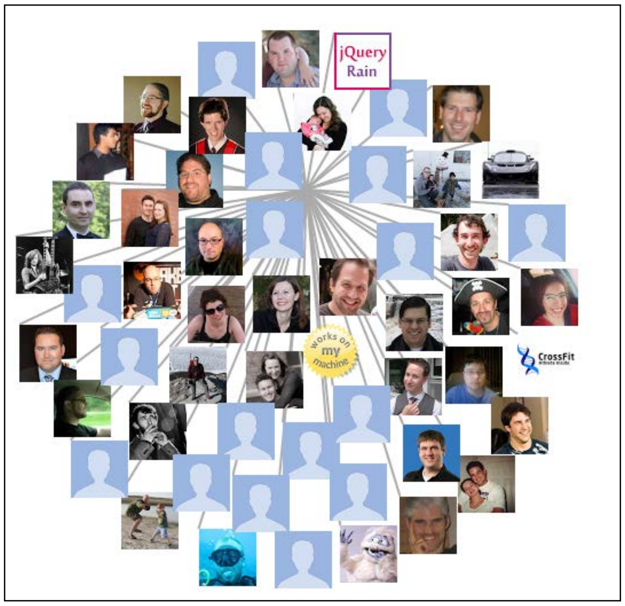
The nodes are a bit more complicated, because for each one we need to set a picture taken from the Google+ data, the initial location, as well as the dimensions. We also need to attach an event handler to the nodes so that when dragged, the `force.drag` action is fired:

```
var node = graph.selectAll(".node")
           .data(nodes)
           .enter()
            .append("image")
             .attr("xlink:href", function(d){ return d.image;})
            .attr("class", "node")
            .attr("r", 15)
            .attr("x", 250)
            .attr("y", 250)
            .attr("width", 50)
            .attr("height", 50)
            .call(force.drag);
```

Finally, we need to instruct `d3` what action should be taken on each tick when animating the graph:

```
force.on("tick", function () {calculatePosition(link, node);});
```

This will result in a graphic that shows my links to friends on Google+, as shown in the following screenshot. If you click and drag a node, it will move and all the nodes will rebalance themselves to account for the movement:



Summary

The `limited` API of Google+ does limit some of the visualizations we can create. There has been a rumor for years that Google will surface additional functionality in Google+, but I have seen no real action so far. You should now be able to authenticate against Google+ and retrieve data from it. You should also be able to make use of the graphically pleasing force-directed layout from `d3` as well as any of the other available layouts.

Index