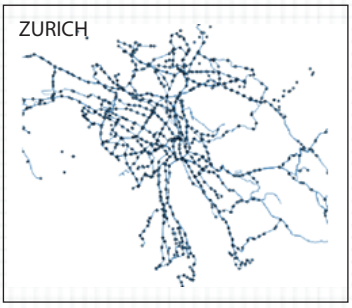
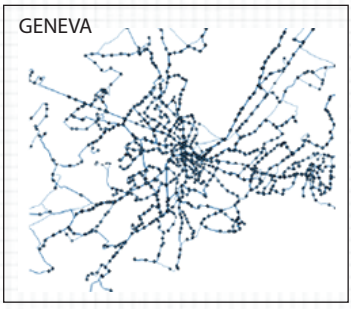
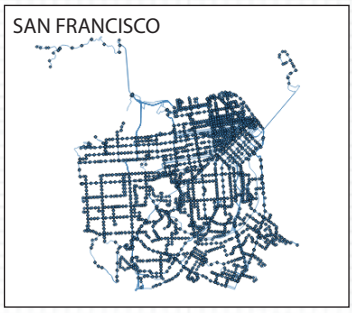
Developing a D3.js Edge:

Constructing reusable D3 components and charts







Chris Viau, Andrew Thornton, Ger Hobbelt, and Roland Dunn

BLEEDINGEDGEPRESS

Developing a D3.js Edge

Published : 2013-06-18
License : MIT

Developing a D3.js Edge

by

Chris Viau, Andrew Thornton, Ger Hobbelt, and Roland Dunn

Bleeding Edge Press, © 2013

ISBN: 978-1-939902-02-3

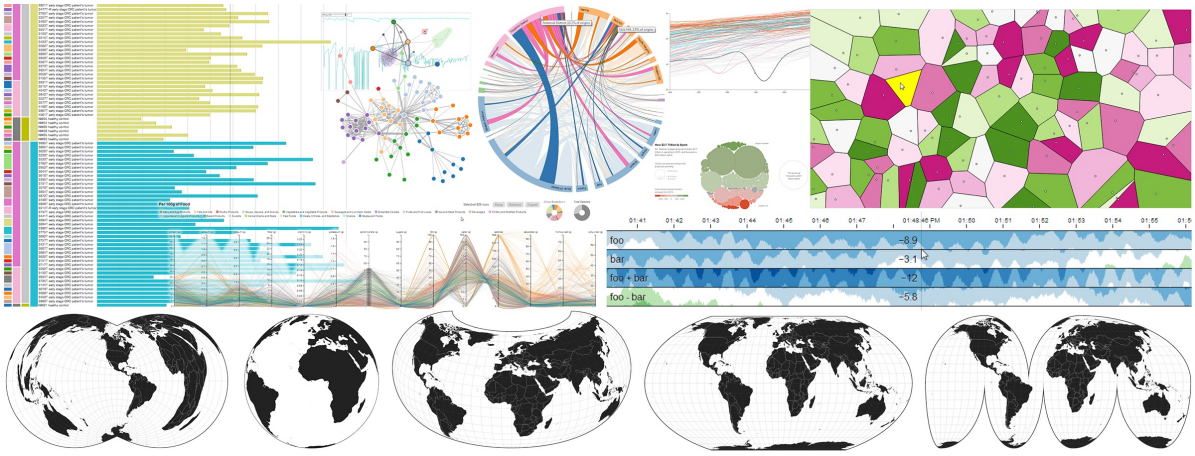
v 1.0

This book is aimed at both intermediate and advanced D3.js developers, particularly those interested in understanding the key modular design patterns at the heart of D3.js, and those wishing to know how to apply them to their own projects.

What is D3?

D3 is a visualization toolkit that Mike Bostock, author of Protovis (http://mbostock.github.com/protovis/), created along with Heer & Ogievetsky. D3.js emerged from Mike's previous work while part of Stanford University's Stanford Visualization Group.

D3.js quickly gathered fame due to its very readable coding patterns and ease of use, combined with an unprecedented flexibility and adaptability. This allows developers of all backgrounds to create both static and *animated* visualizations of data sets ranging from the basic to the impressively complex:



Why should you use D3.js?

There are many reasons for using D3.js:

- It has been architected to facilitate both simple and flexible custom data displays in web pages.
- At its heart, D3.js enables a simple yet powerful mechanism whereby developers can associate data with ('bind data' to) a visual representation of that data (e.g. a chart). This association, however, allows for a significant degree of separation of code.
- D3.js does not restrict you to a limited set of configuration options that other easier visualization libraries offer with their charts.
- D3.js is tailored to facilitate visualization *animation*. You can plot a simple static graph just as easily, but when you want to visually *mutate/transform* that graph, nothing compares to using D3.js.
- D3.js is a library backed by a large community with a wide range of interests. The core team is small, but there's significant activity in the community. Consequently, when you are stuck, chances are high you'll receive help getting unstuck quickly and thoroughly.

TIP Charting libraries built with D3.js, such as NVD3, xChart, Dc.js, Rickshaw and DexCharts, provide excellent wrappers around the default charts in D3, which is very useful when you need to quickly put together a standard chart. For example, NVD3 successfully uses the same reusable pattern we demonstrate throughout this book.

How Does this Book Help?

If you came across the title of this book and decided to read this introduction, more than likely you have some experience with D3.js (if not, head over to d3js.org to learn more). Perhaps you have studied one of the many examples online and you've adapted it to fit your specific needs. You have copied and pasted the code into your favorite text editor, changed the underlying dataset, and tweaked a few variables here and there until you found what you needed. For the most part, this works great. With all the outstanding examples provided by Mike Bostock and the enthusiastic D3.js community, chances are, what you want to visualize has probably been done before, and with some minor modifications, you can get it to work for you as well.

Maybe you developed your own visualization from scratch because you couldn't find exactly what you wanted. Starting from a clean slate, you began coding up your visualizations, figuring it out as you went along, likely ending up with something that resembles spaghetti code. Powering through, you got the visualization exactly the way you wanted and proudly published it for the world to see. And as the world is never one to be shy, somebody spoke up to say, “This visualization would be even better if you would simply change this tiny detail!” You then promptly replied,

"You know what, you're right! Now just let me figure out where I set that in this mess I call code!" Or better yet, perhaps the world is so in love with your visualization that they want to use it too, tweaking it slightly to fit their needs. Now they get the pleasure of diving into your many layers of code to adapt it for their use, surely cursing you along the way. And finally, after the glory surrounding your visualization has faded and you have moved on to new projects, you realize that with some minor modifications, your current project could use that visualization and you won't need to duplicate all that work! So you proudly pull it off the shelf, blow the dust off of it and dive into your many layers of code, surely cursing yourself along the way.

This scenario is rather common, and certainly one that all of the authors of this book have experienced first hand. Fortunately, it is one that can easily be avoided. Having to relearn your own code every time you need to make a modification, or when you want to reuse it is both inefficient and unnecessary. Mike Bostock introduced a reusable pattern to avoid exactly these scenarios, and it is this pattern that we will expand on in this book.

By demonstrating this pattern through a real world example, we intend to show you the benefits of using a reusable API for your visualizations, and how to develop your own reusable APIs to fit your specific needs. By following this approach, you can begin building your own library of modules that you can pick and choose from, and with simple getter and setter functions, quickly adapt them for your current needs. In doing so, your code reuse will increase, replacing your spaghetti code with a much clearer modular pattern. This pattern is not only easier to read and understand, it's also easier to write, and just as importantly, test! All of this will help you to develop a D3.js edge!

How Was this Book Produced?

The book was created through a mini virtual "book sprint". From booksprints.net:

> A Book Sprint brings together a group to produce a book in 3-5 days. There is no pre-production and the group is guided by a facilitator from zero to published book. The books produced are high quality content.

What Do You Need to Know?

To get the most out of this book you need to have at least some experience in JavaScript and with creating graphics using D3.js. It is also handy to have a good working relationship with your debugger (Chrome Developer panel or the Firefox/Firebug combo). Although we do not address those tools directly in this book, we believe that stepping through the examples provided with this book will enhance your understanding of the concepts explained and showcased here. We assume that you're somewhat comfortable with any and all of these concepts:

- Using D3.js
- *Functions* as first-class citizens
- *Closures*
- Some general terminology (*methods*, object *properties*, *private* vs. *public, function chaining, etc.)*

Source Code: Availability & Organization

All the source code referred to in this book is available at github: https://github.com/backstopmedia/D3Edge
You can either download the latest version of all the files in a ZIP archive at the URL below or use git to fetch the repository: https://github.com/backstopmedia/D3Edge/archive/master.zip.

The source files are organized as follows:

- Unless otherwise noted in the chapters themselves, the sources for each chapter are stored in the `code/<chapter>` directory, where `<chapter>` denotes the chapter number and title, e.g. `code/Chapter01/`
- Files that are shared by many reside in another part of the directory hierarchy:
  - JavaScript libraries such as D3.js are located in the `lib/` directory tree
  - Data files are stored in the `data/` directory tree

The Authors

Chris Viau is a PhD in Computer Engineering from Ecole de Technologie Superieure in Montreal, specialized in Information Visualization. He lives between San Francisco and Montreal, working on visual analytics for BigData. You can easily find him on the D3 Google Groups, follow his D3 Twitter feed @d3visualization, or meet him at the Bay Area D3 User Group.

Andrew Thornton is our Austin, TX, USA based data specialist and one of the code artists responsible for the D3.js application demo software and a couple of utilities to make our work with this bleeding edge way of producing technology books through BookType the best possible experience. Andrew has, next to handling the nasty job of wading through the raw data muck that we call data sources, contributed the flesh and bone (and quite a bit of skin as well) of the core of our book. He can be found on Twitter, @graftdata, in the D3 Google Groups, and as the leader of the Austin d3.js Meetup group.

Roland Dunn is a data viz and optimization consultant based in London, UK with over a decade of working in digital media. Roland has contributed to a number of chapters in the book, while also helping keep the team on a clear and (hopefully) coherent narrative. Roland is a partner at http://www.refinedpractice.com/, has his own experimental site at http://www.cloudshapes.co.uk/ (which includes a couple of examples following some of the techniques described in this book), is on Twitter at http://twitter.com/roland_dunn/, and can sometimes be found at

the London D3 Meetup.

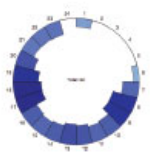
Ger Hobbelt, Dutch national and citizen of political centre The Hague has over two decades of professional IT experience, ranging from 'hard real-time' embedded development all the way to mission-critical financial data processing application development and data visualization. His specialties are risk analysis and modeling / architecture. Thanks to his personal makeup and love for all things fundamental, he never stopped to ask 'why?' As a former trainer for fundamental process and organization analysis, it was rather obvious he be assigned the task to write the bits addressing the theoretical and conceptual fundaments underpinning solid D3 software development, and maybe a few twiddled bits here and there.

Troy Mott is the publisher (bleedingedgepress.com)of this book and contributor through providing most of the infrastructure required to facilitate a global team of authors and developers in producing this technology book.

Next Chapter

In the next chapter, we dive into the standard approaches to using D3.js, how D3.js lends itself to re-usability, and some of the issues that you can encounter when pushing these approaches to the limits.

1. Standard D3



IN THIS CHAPTER

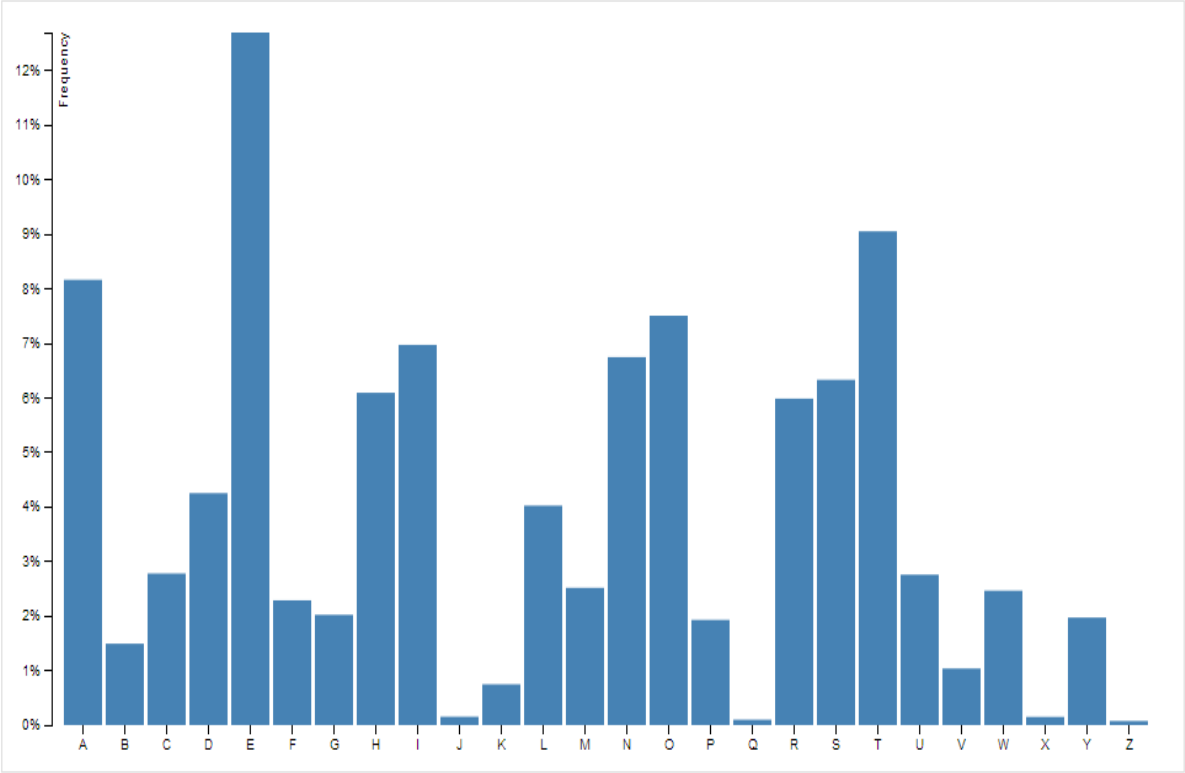
- Introduction to an example of a typical D3 chart
- Explanation of the key D3 elements used
- Illustration of issues arising from re-using a standard chart

---

If you work through the D3.js tutorials and the D3.js examples, you'll eventually become familiar with a typical way of using D3.js. By this we mean with a set of techniques, and code structures that most people frequently use when working with D3.js. These techniques and code strucutures work great for demonstrating the core concepts of D3.js to those just learning and implementing one-off visualizations. However, as it is their main focus to demonstrate core concepts, they often don't address the problem of creating more than one visualization efficiently. Somebody just learning D3.js would most likely be tempted to simply copy and paste the code used to create a visualization when they need another one that uses new data, perhaps unware that there is a better way. A way that is really the essence of the D3.js. Before we demonstrate this better way, let's first take a look of this "typical" standard D3 use, and some of the pitfalls of re-use following this pattern.

A Typical D3 Chart

Rather than re-invent the wheel, we've taken a typical example of how to use D3.js, and provided a walkthrough of its constituent parts and how they all relate to one another:



The above is a simple bar chart illustrating the frequency that letters of the alphabet appear in some text.

NOTE code/Chapter01/TypicalBarChart/

The data is a simple .tsv (Tab Separated Values) file, with one column for letters and one column for the frequency of occurrence of the letters in the text:

```
001: letter frequency
002: A .08167
003: B .01492
004: C .02780
005: D .04253
```

```
006: E .12702
```

The source code that generates the above chart is listed in full next (we walk through the entire code right after this source code):

```
001: var margin = {top: 20, right: 20, bottom: 30, left: 40},
002:         width = 960 - margin.left - margin.right,
003:         height = 500 - margin.top - margin.bottom;
004:
005:     var formatPercent = d3.format(".0%");
006:
007:     var x = d3.scale.ordinal()
008:         .rangeRoundBands([0, width], .1);
009:
010:     var y = d3.scale.linear()
011:         .range([height, 0]);
012:
013:     var xAxis = d3.svg.axis()
014:         .scale(x)
015:         .orient("bottom");
016:
017:     var yAxis = d3.svg.axis()
018:         .scale(y)
019:         .orient("left")
020:         .tickFormat(formatPercent);
021:
022:     var svg = d3.select("#figure").append("svg")
023:         .attr("width", width + margin.left + margin.right)
024:         .attr("height", height + margin.top + margin.bottom)
025:         .append("g")
026:         .attr("transform", "translate(" + margin.left + "," + margin.top + ")");
027:
028:     d3.tsv("data.tsv", function(error, data) {
029:
030:         data.forEach(function(d) {
031:             d.frequency = +d.frequency;
032:         });
033:
034:         x.domain(data.map(function(d) { return d.letter; }));
035:         y.domain([0, d3.max(data, function(d) { return d.frequency; })]);
036:
037:         svg.append("g")
038:             .attr("class", "x axis")
039:             .attr("transform", "translate(0," + height + ")")
040:             .call(xAxis);
041:
042:         svg.append("g")
043:             .attr("class", "y axis")
044:             .call(yAxis)
045:             .append("text")
046:             .attr("transform", "rotate(-90)")
047:             .attr("y", 6)
048:             .attr("dy", ".71em")
049:             .style("text-anchor", "end")
050:             .text("Frequency");
051:
052:         svg.selectAll(".bar")
053:             .data(data)
054:             .enter().append("rect")
055:             .attr("class", "bar")
056:             .attr("x", function(d) { return x(d.letter); })
057:             .attr("width", x.rangeBand())
058:             .attr("y", function(d) { return y(d.frequency); })
059:             .attr("height", function(d) { return height - y(d.frequency); });
```

The first significant chunk of code sets some chart attributes and builds a scale using a reusable scale function and an axis using one of the most important reusable components of the D3.js core (d3.axis).

Walkthrough of the Code

1. A margin object (see Mike Bostocks 'conventional margins' approach to margins) is set up, plus the resulting width and height of the final chart, which uses the values defined in the margin object:

```
var margin = {
        top: 20,
        right: 20,
        bottom: 30,
        left: 40
    },
    width = 960 - margin.left - margin.right,
    height = 500 - margin.top - margin.bottom;
```

2. The D3.js format function generates a *format function*, which in turn is used later on to format the percentages as easy-on-the-eye human readable percentages on the y-axis. D3.js often uses this pattern; a configurable function returning a function to be used on the data:

```
var formatPercent = d3.format(".0%");
```

3. The ordinal scale function is used to create an ordinal scale object for the x-axis (i.e. the letters). The code also defines the output range of the ordinal scale object, between zero and width, using the rangeRoundBands attribute.

   Note that at this stage the x ordinal scale object has not yet been provided with any input *domain,* i.e. it doesn't yet know what it is mapping *from*. This bit only defines what it is mapping *to*.

   The rangeRoundBands attribute tells the scale to divide the range of output values into blocks, or bands, based upon the number of values in in the input domain.

```
var x = d3.scale.ordinal()
    .rangeRoundBands([0, width], .1);
```

4. The next bit of code creates a linear scale object to be used for the y-axis (i.e. to visually represent the frequency-of-occurrence percentage associated with each letter). No input domain is defined at this point, since the domain is defined later on in the code.

   Note also that the output range of the y scale is from height to zero, not from zero to height. Each bar is drawn as a rectangle, specifying the x,y of the rectangle (top-left of the rectangle), and the width and the height.

   The SVG co-ordinate system has x-axis values increasing horizontally from left to right, and y-axis values increasing from top to bottom of the screen; whereas the co-ordinate system of the chart we wish to render has y-axis values increasing vertically up the screen.

   So, the scale to be used for the y-axis is inverted, i.e. from height to zero.

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

5. Next, the code uses the d3.svg.axis components to create an axis object for the x-axis. The scale attribute of the axis object is set to the ordinal scale we created above.

   The orient attribute is interesting and worth a little note. Later on in the code a g SVG element is created to render (i.e. display) the actual x-axis. This g element is shifted to the bottom of the chart (height pixels down the page) using a transform call. The orient attribute of the xAxis object states how the xAxis object will be positioned **relative** to this g element. In this case it's at the bottom of the g element:

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

6. Next, the code creates a yAxis object for the y-axis, orientates the yAxis object to the left of the g element created for the y-axis, and passes the *format function* created earlier to format the percentage values on the y-axis:

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

7. The next bit of code creates the main SVG element in which the bar chart will itself be rendered, specifying its width and height. It appends a g element, and shifts—using the transform and translate functions—the g element down and right using values from the margin object:

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

8. Now we get to the code that actually starts to use the data.

   The d3.tsv component loads up the data from the data.tsv file and calls the anonymous callback function that starts processing the data:

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

9. Remember that the x and y variables are the ordinal and linear scale objects, and are used to provide the xAxis and yAxis objects with the data that those axis objects will actually render. Here the code defines the actual input data for those scale objects.

   First up the code uses the Array.map method to create an array of just the letters themselves from the data. This array of letters is then used to define the ordinal input domain of the x scale object:

```
x.domain(data.map(function(d) { return d.letter; }));
```

   Second, the code defines the input values (i.e. the letter frequency values) to be used in the y linear scale object. The y linear scale object then maps from those input values to the pixel range specified earlier when the y linear scale object was declared. Remembering how the y scale object is defined earlier on, y(0) will return a value of 450 (height), and y(0.12702) will return zero (0.12702 being the maximum frequency):

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

10. Next up, the code creates a g element in which the visual elements of the x-axis will be rendered. The g object is assigned the CSS class ("x axis"), and is shifted down the page height pixels, and finishes off with a JavaScript call to the xAxis object. This renders the actual visual x-axis within this g element:

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

11. Similar code follows for the actual visual y-axis, except with a little more going on. First up, the g element used to render the y-axis is created, and the yAxis object is called to render the actual visual y-axis within this g element.

    Note that the actual g element is **not** transformed and shifted; however don't forget that when the yAxis was created, the orientation was specified as "left." So, there was actually no need to transform the actual g element.

    The code then appends a text element, transforms this text element ninety degrees anti-clockwise, shifts it a few pixels to the right (using the y and dy attributes), aligns the actual text of the text element, and finally specifies the actual text used within the text element. The dy attribute is being used to transform the text so that the rotation point is on the baseline of the typeface, instead of being at the topmost point of the typeface.

```
svg.append("g")
    .attr("class", "y axis")
    .call(yAxis)
    .append("text")
    .attr("transform", "rotate(-90)")
    .attr("y", 6)
    .attr("dy", ".71em")
    .style("text-anchor", "end")
    .text("Frequency");
```

12. The last chunk of code is the actual creation of the "bars" in the bar chart. It's in this block of code where we come across an example of the *enter/update/exit pattern* that you will have encountered in D3.js tutorials and examples.

    The selectAll attempts to select all SVG elements with the CSS "bar" class - but there aren't any. The .data() method specifies which data the following code will be applied to, which in this case is the contents of the .tsv file. The enter states that for any "bar" elements not yet created (which in this case is all of them), append an SVG rect element.

    For each SVG rect element, assign the bar CSS class and assign the x co-ordinate by mapping from letter to

pixel using the x-scale. You then specify the width again using the x-scale, assigning the y coordinate and the height using the y-scale.

```
svg.selectAll(".bar")
    .data(data)
    .enter().append("rect")
        .attr("class", "bar")
        .attr("x", function(d) { return x(d.letter); })
        .attr("width", x.rangeBand())
        .attr("y", function(d) { return y(d.frequency); })
        .attr("height", function(d) { return height - y(d.frequency); });
```

That's it! This is a typical D3 chart.

Creating Multiple Instances of the Chart

Now, how about if you want to use the above code to create more than one bar chart on the same page? Chances are, your first instinct would be to simply copy and paste the code, and modify the data. Take a look at code/Chapter01/TwoBarCharts. This takes a simplified version of the above bar chart and effectively copies and pastes the code to create two charts.

Here is a glimpse at the code to make two charts:

```
var data1 = [10, 20, 30, 40];

var w = 400,
    h = 300,
    margins = {left:50, top:50, right:50, bottom: 50},
    x1 = d3.scale.ordinal().rangeBands([0, w]).domain(data1);
    y1 = d3.scale.linear().range([h,0]).domain([0, d3.max(data1)]);

var chart1 = d3.select("#container1").append("svg")
      .attr('class', 'chart1')
      .attr('w', w)
      .attr('h', h)
    .append('g')
      .attr('transform', 'translate(' + margins.left + ',' + margins.top + ')');

  chart1.selectAll(".bar")
      .data(data1)
    .enter().append("rect")
      .attr('class', 'bar')
      .attr('x', function (d, i) {return  x1(d);})
      .attr('y', function (d) {return y1(d);})
      .attr('height', function (d) {return h-y1(d);})
      .attr('width', x1.rangeBand())
      .on('mouseover', function (d,i) {
        d3.selectAll('text').remove();
        chart1.append('text')
          .text(d)
          .attr('x', function () {return  x1(d) + (x1.rangeBand()/2);})
          .attr('y', function () {return y1(d)- 5;})
      });

var data2 = [100, 259, 332, 435, 905, 429];

var w = 400,
    h = 300,
    margins = {left:50, top:50, right:50, bottom: 50},
    x2 = d3.scale.ordinal().rangeBands([0, w]).domain(data2);
    y2 = d3.scale.linear().range([h,0]).domain([0, d3.max(data2)]);

var chart2 = d3.select("#container2").append("svg")
      .attr('class', 'chart2')
      .attr('w', w)
      .attr('h', h)
    .append('g')
      .attr('transform', 'translate(' + margins.left + ',' + margins.top + ')');

  chart2.selectAll(".bar")
      .data(data2)
    .enter().append("rect")
      .attr('class', 'bar')
      .attr('x', function (d, i) {return  x2(d);})
      .attr('y', function (d) {return y2(d);})
      .attr('height', function (d) {return h-y2(d);})
      .attr('width', x2.rangeBand())
      .on('mouseover', function (d,i) {
        d3.selectAll('text').remove();
        chart2.append('text')
          .text(d)
          .attr('x', function () {return  x2(d) + (x2.rangeBand()/2);})
          .attr('y', function () {return y2(d)- 5;})
      });
```

As you can see, there is a lot of repetition that goes into making these two charts. Actually, the only difference in the code for each chart, is the data that is used to generate them. It seems rather inefficient to copy and paste code whenever you want to a new instance of a chart. What would happen if we wanted 10 charts, each using a different data set. That would be a lot of duplicated code!

Reuse Or Not To Reuse

The above two-chart code does work as it should, so you may well wonder what's the problem? If you were just creating this one page, with just these two bar charts, and you were never going to create D3.js bar charts again, then this approach is probably OK.

However, imagine that you then wished to create another bar chart on a different page. And then imagine that you wished to change something, for example the layout, or the data-format changed. You would then suddenly have three separate blobs of code to update: three places where you can make three different sets of mistakes.

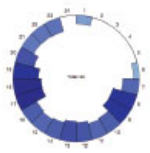
So the question is: how to take this standard approach to creating a D3.js chart—which clearly works—and make it more usable, more straightforward to maintain, and more suitable for sharing with a wider audience?

These are questions that we look to answer in the following chapters.

Summary

Clearly the classic approach to using D3.js works, but we've seen how such an approach doesn't lend itself to elegant, straightforward, easily maintainable, and resuable code.

2. API Requirements



IN THIS CHAPTER

- Definition of an API
- Different meaning of the world "modular"
- Requirements for our API
- Some implementation details

Most D3.js examples show how to build a single standalone chart. In the previous chapter we saw an example of a typical D3.js chart and saw an illustration of some of the issues that can be encountered when we look to reuse a chart multiple times on the same page. In a real-world application, these charts need to interact with a lot of different technologies.

In this chapter, we will see what the requirements are for integrating D3.js charts in an application, and how the reusable API is a good fit for the task.

What's an API?

An Application Programming Interface (API) is "a specification of how some software components should interact with each other." For example, the code can be built as multiple modules exposing some public functions that the other modules can use, as part of a contract between them. We will refine this terminology to better explain the requirements for our API in the contect of D3.js.

For chart code to be reusable, we need to easily instantiate a new chart independent from the other instances. We don't want the properties of one chart propgating to other instances. For this requirement, we will use a chart generator, which is a function returning a new chart object with the configuration we want (for example, a function returning a function). This will ensure that the context of all the settings and methods are relative to the instance of the chart, rather than all charts.

To set and modify individual chart settings, we will use getter and setter functions instead of accessing inner variables directly. Accessing configuration variables through a function has multiple advantages. First, the function can validate the arguments, change the type, compute some side effects, and so on. Second, the inner variable can change name or even be removed, but the getter and setters functions will stay consistently the same. That's the best example of what an API is all about: the interface with your code is consistent and predictable even if the internal implementation is deeply refactored. This ensures that any applications built using your API, including your own applications, don't break if you modify the internals of your code.

Refactoring would be very difficult without a good API. Professional code always uses some kind of unit testing to insure that the API is clear with integration tests, and to make sure that the interaction is constantly respected between modules. The unit tests are mainly a set of expectations on the behavior of a module--of a unit of code. It's easier to aggressively refactor the implementation of a module when the only thing that can't change is the API. The test suite will then quickly identify the part of the code that breaks the contract.

Each chart should be self-contained and decoupled from the others. For example, a chart can ask for a dataset in a specific format without needing to know about the file format coming out of a data manager module instead of having a data manager hardcoded in it. Good modules clearly separate the functionalities. In our example, the chart module takes care of building the chart and rendering it while the data manager handles the dataset request and persistence. If it depends upon other modules, the dependency could be injected instead of hardcoded. The separation of concern between a model, a view and a controller is a well-known pattern. An MV* library is often used for adding this structure to a whole application, handling dependencies (require.js), and communication between modules (AngularJS two-way bindings), and so on. Integrating a D3.js chart in an application is often a matter of building a self contained D3.js chart module that will play nice with all of the other components.

D3.js modular API

In this book, we are mainly concerned with D3.js modules interacting with other D3.js modules. D3.js modules can be of different types. You can build your own chart library as a collection of chart modules. But other pieces of code can be wrapped under a module. We will call a "component" a reusable piece of code generating graphics, which doesn't represent a complete chart, but a part meant to be composed with other elements. D3.js is built out of components. You use these components everytime you generate a visualization with D3.js! The best example is d3.svg.axis. You can draw an axis, but they will be more useful as part of a chart. Components are building blocks, and they are a good illustration of how D3.js will prefer composition to inheritance.

Other modules don't generate graphics. For example, "layouts" like d3.layout are preparing data for the graphics space, and injecting some abstract position and size information that is used to map from data space to pixel space.

Other modules, called "generators," take some data as input and return an SVG string as output. These strings are not graphics element until they are used as SVG attributes and bound to the DOM. Some more modules take care of the non-graphical stuff, like data helpers. d3.nest or d3.extent are good examples of D3 helpers.

All of these packages can be shared as plugins, and are modules you can use to add functionalities to the core functions. Most plugins you will find on the web are complete charts, with a lot of them using the Reusable API. But some non-graphics plugins are also available, like keybinding, graph data manipulation or svg transform. Plugins are a very important part of the D3.js ecosystem. The core D3.js code will probably not expand a lot, in fact, it may even tend to shrink. After all, D3.js is "a visualization kernel" and some plugins and chart libraries are growing in popularity. One of the best examples is the way the map plugin is growing--counting more than 80 projections at the time of writing.

Implementation Overview

Multiple modules can live under a common namespace, to be clearly identified as related, and to expose a single variable to the global namespace. For example, all D3.js code is under the d3 namespace (i.e., d3.select, or d3.svg.axis). In this book, we will add our own namespace to the d3 one:

```
001: d3.edge = {};
```

A module should be self-contained and expose a public API. One way for encapsulating the code is to wrap it inside a simple function:

```
001: d3.edge.simpleChart = function(){ /*chart code*/ };
```

The Reusable API uses a neat trick to expose public functions while keeping others private, using an outside function just for exporting an inner function, and using getters and setters to give access to "private" variables that are in the closure:

```
001: d3.edge.simpleChart = function(){
002:     var height = 100;
003:     function exports(){
004:         /*chart code*/
005:     }
006:  exports.width = function(_x) {
007:  width = _x;
008:  return this;
009:     };
010:  return exports;
011: };
```

This snippet of code is pretty dense and is the core concept of the Reusable API that we will explain and illustrate throughout this book. More specifically, we want to focus on how this pattern helps to fulfill our API requirement. So let's start by listing these requirements.

API requirements

To summarize, we want a modular API. But what does it mean in our own words?

Namespaced: only a single object is exposed to the global scope, preventing name clashing and clearly identifying relationships.

Encapsulated: one way of encapsulating code is to wrap a function around it (simple, IIFE, etc.) to hide a certain amount of code under a simple abstract syntax (i.e., d3.edge.pieChart()).

Decoupled: a module doesn't know about the others. It only handles it's own internal state and behavior, exposing an API for others to use. Dependencies can be handled in multiple ways (i.e., dependency injection).

Consistent: the API never changes, the pattern is clear and the naming is significant.

Composable: preferring composition to inheritance can be really hard when you come from an Object Oriented (OO) background. You can find plenty of resources and debates on the web about prototypal inheritance and other features. Let's just say here that studying D3.js source code is a good way to convince yourself that the functional aspect of Javascript can be pretty powerful. In D3.js, you don't write a "chart" code and then derive a "bar chart" from it. The typical bar chart rather is a composition: an assembly of rectangles, texts, axes, and other modules.

The Reusable API also has some more interesting characteristics. For example, when you update a d3.svg.axis configuration, or when you bind new data to your selection, you simply call the axis again without having to explicitly call a draw or an update method.

```
001: svg.append("g")
002:     .attr("class", "x axis")
003:     .attr("transform", "translate(0," + height + ")")
004:     .call(xAxis);
005:
006: //Update
007: x.domain(d3.extent(data));
008: d3.select(".x.axis")
009:     .call(xAxis)
```

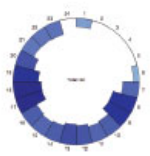
Another interesting feature is that setters also act as getters when no arguments are provided. D3.js also provides

interesting ways of exposing events using d3.dispatch and d3.rebind. There is a lot to learn from D3.js's internal structure. Learning the Reusable API is a good pretext to do so.

Summary

The Reusable API is a variant of a module pattern. Writing modular code can mean a lot of different things. We tried to list some features a modular API should have. With this list of requirements and at least an approximate terminology, you are ready to start learning the Reusable API, how to implement and test it, and use it in a real-world example of integrating D3.js charts in an application.

3. The Reusable API



IN THIS CHAPTER

- A walkthrough of a "Hello World" reusable component
- A discussion of namespace, reusable module, closure, getters and setters

---

In the previous chapter we defined what the words *module* and *plugin* mean, at least in the context of JavaScript, D3.js and this book. We also defined what we mean when we say something is a reusable *function*, *component* or *chart*. We also referred to the Best Practices for D3.js as described by Mike Bostock in "Towards Reusable Charts". The principles outlined in that post are used in the D3.js core as well.

In this chapter we take a look at the implementation of these concepts in the form of a very simple "Hello World" *plugin*.

Hello World

We will use a very simple "Hello world" plugin that just adds a with text that you can hover over with the mouse to see a message in the console. In later chapters you will see the API at work in more detail--for example, how to generate animated transitions and test using a unit test suite.

The next chapter has a complete real-world example to illustrate this pattern in practice, using the bar chart from Chapter 1 ("Standard D3").

NOTE Access the source code: code/Chapter03/

Here is the complete code we will describe. If you are already familiar with this pattern, feel free to jump directly to the line you want to learn more about:

```
001: d3.edge = {};
002:
003: d3.edge.table = function module() {
004:     var fontSize = 10,
005:         fontColor = 'red';
006:
007:     // To get events out of the module
008:     // we use d3.dispatch, declaring a "hover" event
009:     var dispatch = d3.dispatch('customHover');
010:     function exports(_selection) {
011:         _selection.each(function(_data) {
012:             d3.select(this)
013:             .append('div')
014:             .style({
015:                 'font-size': fontSize + 'px',
016:                 color: fontColor
017:             })
018:             .html('Hello World: ' + _data)
019:             // we trigger the "customHover" event which will receive
020:             // the usual "d" and "i" arguments as it is equivalent to:
021:             //   .on('mouseover', function(d, i) {
022:             //       return dispatch.customHover(d, i);
023:             //   });
024:             .on('mouseover', dispatch.customHover);
025:         });
026:     }
027:     exports.fontSize = function(_x) {
028:         if (!arguments.length) return fontSize;
029:         fontSize = _x;
030:         return this;
031:     };
032:     exports.fontColor = function(_x) {
033:         if (!arguments.length) return fontColor;
034:         fontColor = _x;
035:         return this;
036:     };
037:     // We can rebind the custom events to the "exports" function
038:     // so it's available under the typical "on" method
039:     d3.rebind(exports, dispatch, "on");
040:     return exports;
041: };
042:
043: // Setters can also be chained directly to the returned function
044: var table = d3.edge.table().fontSize('20').fontColor('green');
045: // We bind a listener function to the custom event
046: table.on('customHover', function(d, i){
047:     console.log('customHover: ' + d, i);
048: });
049:
050: d3.select('body')
051: .datum(dataset)
052: .call(table);
```

First, we add our sub-namespace to the d3 namespace. It is not essential to do this, but name-spacing is good practice to not pollute the global space.

```
d3.edge = {};
```

We add the table module, which is a simple function returning a function. The outer function serves as the scoped *closure* for our module.

```
d3.edge.table = function module() {
    function exports() {
        //...
    }
    return exports;
};
```

Some variables are available in the closure and not accessible from the outside (private). They have default values.

```
d3.edge.table = function module() {
    var fontSize = 10,
        fontColor = 'red';

    function exports() {
        //...
    }
    return exports;
};
```

In JavaScript, a function is also an object, so we can attach some properties and methods to it. The functional aspect of JavaScript is very powerful, but it forces you to un-learn some habits from your OOP (Object Oriented Programming) background.

```
d3.edge.table = function module() {
    var fontSize = 10,
        fontColor = 'red';
    function exports(_selection) {
        //...
    }
    exports.fontSize = function(_x) {
        //...
    };
    exports.fontColor = function(_x) {
        //...
    };
    return exports;
};
```

These "public" functions will be used as *getters* and *setters* at the same time. They are getters when no argument is passed to the function; otherwise they set the private variable to the new value passed as an argument. When setting, we return the current context `this`, as we want these methods to be chainable.

```
exports.fontSize = function(_x) {
    if (!arguments.length) return fontSize;
    fontSize = _x;
    return this;
};
```

Let's now illustrate some less basic features. One way for the module to expose events to the outside world is by using an implementation of the pubsub (Publish/Subscribe) pattern. We use `d3.dispatch` to declare an *event* that we can then listen to from the outside when it's triggered in the module.

```
//Declare
var dispatch = d3.dispatch('customHover');

//Trigger
dispatch.customHover()
// Bind to
.on('customHover', function(){ /* user code ... */ });
```

For the event to be accessible from the outside, it needs to be bound to the module itself. We use `d3.rebind` for this task, rebinding the event to the "`on`" method (following D3.js convention).

```
d3.rebind(exports, dispatch, "on");
```

We now have a complete module. Now we need to build some basic HTML and attach it to our page. A D3.js selection will be passed to the function as the parent of our generated html. `_selection.each` loops through the selected elements and the `_data` object attached to each DOM element will be used as the text of our generated `<div>`. Some styling is using the private properties. That's basically the place where you put your standard D3.js code.

```
function exports(_selection) {
    _selection.each(function(_data) {
        d3.select(this)
        .append('div')
        .style({
            'font-size': fontSize + 'px',
            color: fontColor
        })
        .html('Hello World: ' + _data)
        .on('mouseover', dispatch.customHover);
    });
}
```

That's it for the module. The way to use it is to first instantiate the chart, then pass attributes and bind events.

```
var table = d3.edge.table().fontSize('20').fontColor('green');
table.on('customHover', function(d, i){
    console.log('customHover: ' + d, i);
});
```

The module will come into action once you pass your D3.js selection to be used as the parent DOM element.

```
d3.select('#figure')
.datum(dataset)
.call(table);
```

Using the D3.js selection.call function is equivalent do using it this way:

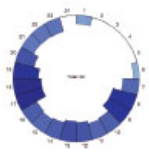
```
var parent = d3.select('#figure').datum(dataset);
table(parent);
```

That's it!

Summary

We've shown the barebones of how to write a reusable D3.js plugin. In the next chapter we expand the complexity of the examples and show how to create multiple instances of the same plugin.

4. Reusable Bar Chart



IN THIS CHAPTER

- Examples of reusable bar charts using the reusable plugin API
- A discussion of the customHover event and generation of axis
- Explore binding data to the bar charts
- Cover flexible methods of creating the charts

So far we've discussed the plugin API and we've shown a basic "Hello World" example of a module. In this chapter we take an enhanced version of the bar chart that we showed in Chapter 1 ("Standard D3") and illustrate how we can re-write this using the plugin API.

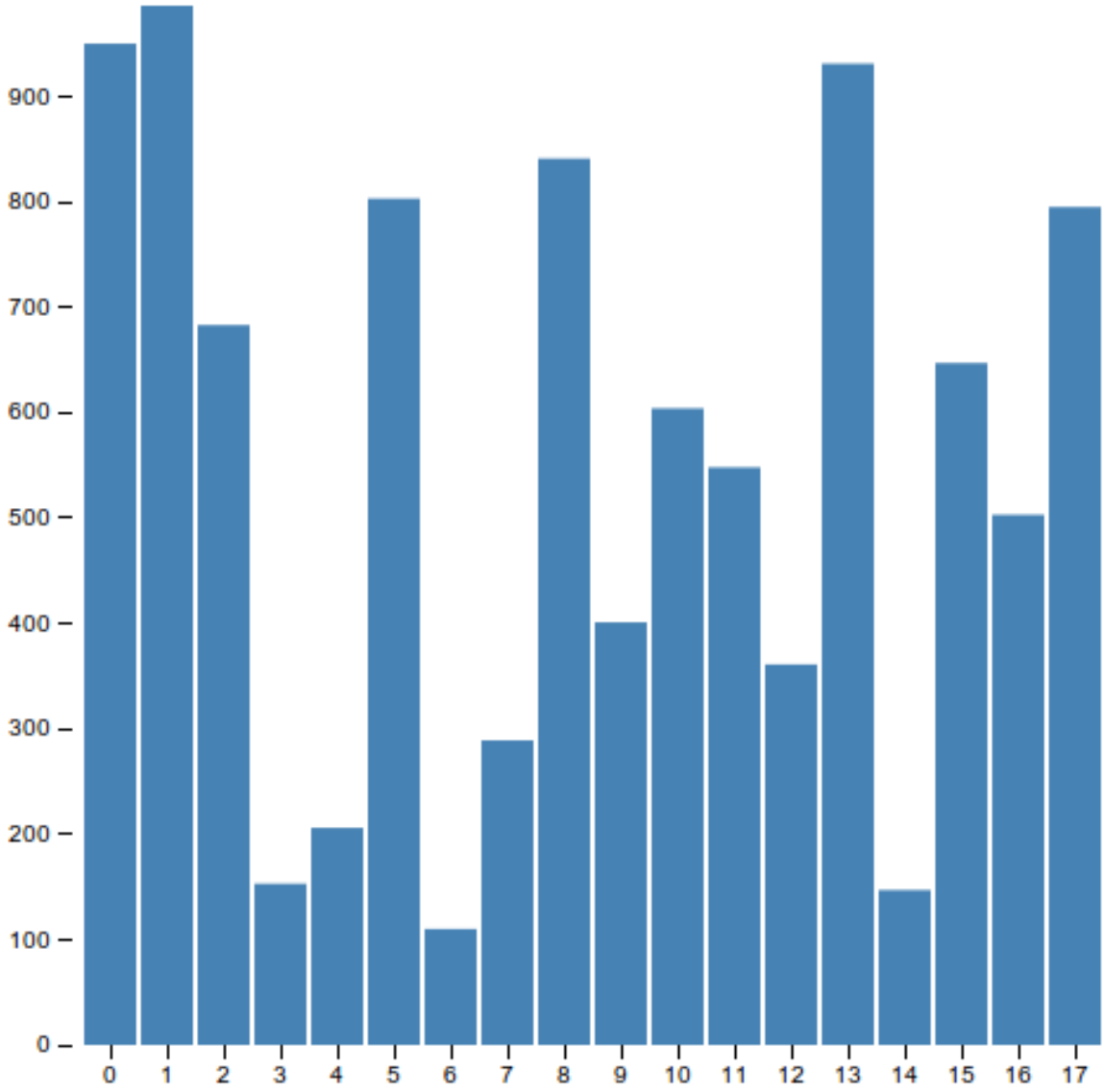
At the end of this chapter we list how you can quickly amend this sample code to have three separate instances of the bar chart module all on the same page.

We also provide a walkthrough of the "Bar Chart with Axes" example to illustrate how to create a bar chart plugin.

NOTE Access the source code: code/Chapter04/ReusableWithAxes/

Description of The "Reusable Bar Chart with Axes" Example

If you run the example code in code/Chapter04/ResuableWithAxes/, you should see the following (or something like the following), which should randomly change every few seconds:



Before diving into the actual code, let's clarify what's going on in terms of SVG elements:

- One <svg> element per chart that contains the following code.
- One SVG <g> element that contains the chart.
- One SVG <g> element for each of the bars, the x-axis and the y-axis.
- <g> elements for each entry on the x and y axis.

This structure is created below. Rather than manually recreate this structure for each instance of our chart, we can let our resuable pattern take care of that for us! Let's walkthrough the code to make this happen.

```
// Main SVG element. One per instance of the chart module:
<svg class="chart" height="500" width="500">
// Main containing SVG 'g' element
<g class="container-group">
  // SVG 'g' element to contain the actual bar's of the bar chart.
  <g class="chart-group">
    <rect class="bar" x="9" width="8" etc. ...
    <rect class="bar" x="18" width="8" etc. ...
  </g>

  // SVG 'g' to contain the x-axis
  <g class="x-axis-group axis">
    <g style="opacity: 1;" transform="translate(13,0)">
      <line class="tick" y2="6" x2="0"></line>
      <text y="9" dy=".71em" style="text-anchor: middle;" x="0">0</text>
    </g>
    <g style="opacity: 1;" transform="translate(22,0)">
      <line class="tick" y2="6" x2="0"></line>
      <text y="9" dy=".71em" style="text-anchor: middle;" x="0">1</text>
    </g>
  </g>

  // SVG 'g' to contain the y-axis
  <g class="y-axis-group axis">
    <g style="opacity: 1;">
      <line class="tick" x2="-6" y2="0"></line>
      <text x="-9" dy=".32em" style="text-anchor: end;" y="0">0</text>
    </g>
    <g style="opacity: 1;" transform="translate(0,394.8254699707031)">
      <line class="tick" x2="-6" y2="0"></line>
      <text x="-9" dy=".32em" style="text-anchor: end;" y="0">100</text>
```

```
    </g>
  </g>
</g>
</svg>
```

Code Walkthrough

Soon we will outline the main differences between this example code, and that outlined in the "Hello World" example in the previous chapter.

First up though, just like all the previous examples, let's setup our module in its own namespace:

```
// Setup our barchart in the d3.custom namespace.
d3.custom = {};
```

Start the definition of the module, declare a set of private variables, and declare the dispatcher function for the custom customHover event:

```
d3.custom.barChart = function module() {
    // Various internal, private variables of the module.
    var margin = {top: 20, right: 20, bottom: 40, left: 40},
        width = 500,
        height = 500,
        gap = 0,
        ease = 'bounce'; // Use the 'bounce' transition type.
    var svg;

    // Dispatcher for the 'customHover' event
    var dispatch = d3.dispatch('customHover');
```

Next, move onto the start of the main internal functionality of the module, and begin by defining the visible width and height of the chart:

```
// Main internal module functionality:
function exports(_selection) {
    _selection.each(function(_data) {
        var chartW = width - margin.left - margin.right,
            chartH = height - margin.top - margin.bottom;
```

Here the code defines various functions and components to be used to draw the x and y axis:

```
// x and y axis variables and functions
var x1 = d3.scale.ordinal()
    .domain(_data.map(function(d, i) { return i; }))
    .rangeRoundBands([0, chartW], .1);

var y1 = d3.scale.linear()
    .domain([0, d3.max(_data, function(d, i) { return d; })])
    .range([chartH, 0]);

var xAxis = d3.svg.axis()
    .scale(x1)
    .orient('bottom');

var yAxis = d3.svg.axis()
    .scale(y1)
    .orient('left');

// Define the width of each bar.
var barW = chartW / _data.length;
```

Next the code creates the <g> elements, and glues them together into the structure we described above:

```
// Append the main 'svg' element if it doesn't exist for this instance of the module.
// Apppend the main 'g' elements.
// The 'classed' attributes define the CSS class.
if (!svg) {
    svg = d3.select(this)
        .append('svg')
        .classed('chart', true);
    var container = svg.append('g').classed('container-group', true);
    container.append('g').classed('chart-group', true);
    container.append('g').classed('x-axis-group axis', true);
    container.append('g').classed('y-axis-group axis', true);
}
```

Move the main <svg> element and x and y axis into place.

```
// Transform the main 'svg' and axes into place.
svg.transition().attr({width: width, height: height});
svg.select('.container-group')
    .attr({transform: 'translate(' + margin.left + ',' + margin.top + ')'});

svg.select('.x-axis-group.axis')
    .transition()
    .ease(ease)
    .attr({transform: 'translate(0,' + (chartH) + ')'})
    .call(xAxis);

svg.select('.y-axis-group.axis')
    .transition()
    .ease(ease)
    .call(yAxis);
```

Calculate a couple of variables used to layout the bars. Note that these variables will change every time the "exports" code is called (provided of course the data actually changes).

```
// Couple of variables used to layout the individual bars.
var gapSize = x1.rangeBand() / 100 * gap;
var barW = x1.rangeBand() - gapSize;
```

Next up we move onto the *enter, update & exit* code of the module.

First, select all of the elements with the "bar" class. If there aren't any, create them (this is the classic D3 'enter' section of the code):

```
// Setup the enter, exit and update of the actual bars in the chart.
// Select the bars, and bind the data to the .bar elements.
var bars = svg.select('.chart-group')
    .selectAll('.bar')
    .data(_data);
// If there aren't any bars create them
bars.enter().append('rect')
     .classed('bar', true)
     .attr({x: chartW,
         width: barW,
         y: function(d, i) { return y1(d); },
         height: function(d, i) { return chartH - y1(d); }
     })
     .on('mouseover', dispatch.customHover);
```

If there are updates, apply them using a transition (the 'update'):

```
// If updates required, update using a transition.
bars.transition()
    .ease(ease)
    .attr({
        width: barW,
        x: function(d, i) { return x1(i) + gapSize / 2; },
        y: function(d, i) { return y1(d); },
        height: function(d, i) { return chartH - y1(d); }
    });
```

Finally, if exits need to happen, apply a transition and remove the DOM nodes when the transition has finished (the 'exit'):

```
// If exiting, i.e. deleting, fade using a transition and remove.
bars.exit().transition().style({opacity: 0}).remove();
```

A series of getter/setter functions enhances the plugin API:

```
// Getter/setter functions
exports.width = function(_x) {
    if (!arguments.length) return width;
    width = parseInt(_x);
    return this;
};
exports.height = function(_x) {
    if (!arguments.length) return height;
    height = parseInt(_x);
    return this;
};
exports.gap = function(_x) {
    if (!arguments.length) return gap;
    gap = _x;
    return this;
};
exports.ease = function(_x) {
    if (!arguments.length) return ease;
    ease = _x;
    return this;
};
```

Do the rebinding of the 'customHover' event:

```
// Rebind 'customHover' event to the "exports" function, so it's available "externally" under the typical "on" method:
d3.rebind(exports, dispatch, 'on');
return exports;
```

That's it for the definition of the module. Next, we look at actually creating an instance of a module.

First, create an instance of a bar chart. Neither data nor selection has yet been passed to the chart, so nothing will actually happen based upon this function call.

```
var chart = d3.custom.barChart();
```

The update() function is called every second and creates a random data set, selects the <body> element in the HTML, passes the "data" variable containing the random data and finally makes the chart actually do something (by passing it the selection which the data has been bound to).

```
function update() {
    var data = randomDataset();
    d3.select('#figure')
        .datum(data)
        .call(chart);
}
```

Here's the small helper function to generate a random set of data:

```
// Generate random sets of data.
function randomDataset() {
    return d3.range(~~(Math.random() * 50)).map(function(d, i) {
```

```
        return ~~(Math.random() * 1000);
    });
}
```

The double-tilde (~~) is a double NOT *bitwise operator*. It is used here as a generally faster substitute for Math.floor(). You can check jsperf.com for empirical benchmark test results, or create your own: this particular issue has been tested extensively.

d3.range: Generates an array containing an arithmetic progression, so d3.range(~~(Math.random() * 50)) generates an array of numbers starting at zero, and ending at a random number between 0 and 50, e.g. [0,1,2,3,4,5, ..., 35]. It's basically a quick way of generating an array of between 0 and 50 numbers.

```
.map(function(d, i) {
    return ~~(Math.random() * 1000);
});
```

The above fills each entry in the array with a random value between 0 and 1000.

The code below makes an initial call to update to kick things off, and then periodically (every second) calls the update function:

```
// Call the update function to actually provide data to the charts and render the data.
update();

// Call the update function once per second.
setInterval(update, 1000);
```

Creating Multiple Instances

The above is a walk-through of a barchart module/plugin, which is very handy when you need to create an instance of a barchart. We can quickly illustrate just how re-usable this code actually is.

You can amend the above code by changing the following:

```
var chart = d3.custom.barChart();
var chart2 = d3.custom.barChart();

function update() {
    var data = randomDataset();
    d3.select('body')
        .datum(data)
        .call(chart);

    var data2 = randomDataset();
    d3.select('body')
        .datum(data2)
        .call(chart2);
}
```

You now have two independent instances of the bar-chart, each with their own data, all on the same page, with no spaghetti code.

Flexibility in Creating Plugins

To whet your appetite for more, we have included two more examples of reusable bar chart plugins:

- Remember the bar chart we created in chapter 1 ("Standard D3")? The spitting image of that one, turned into a reusable plugin, is available in code/Chapter04/ResuableBarChart/. (The bar chart plugin we discussed above is a rather enhanced version of that one!)
- A further enhanced version of the reusable chart plugin discussed above is available in code/Chapter04/ResuableBarChartWithTransitions: this one includes animation support for all parameters.

Also note this bit of code in that last example (code/Chapter04/ResuableBarChartWithTransitions/):

```
// Trick to just append the svg skeleton once
var svg = d3.select(this)
    .selectAll("svg")
    .data([_data]);
svg.enter().append("svg")
    .classed("chart", true);
```

Contrast that with this bit, since it is used in the bar chart plugin discussed above:

```
// Append the main 'svg' element if it doesn't exist for this instance of the module.
// Apppend the main 'g' elements.
// The 'classed' attributes define the CSS class.
if (!svg) {
    svg = d3.select(this)
        .append('svg')
        .classed('chart', true);
```

The former way is a smart way to solve the problem where we know *animatable* reusable charts will be called many times in the web page's lifetime without the need to store the original "svg" d3.selection in the closure.

The code snippet .data([_data]) turns our entire data array into a single data entry for that specific d3.selection, which is exactly what we want. We only wish to create a single <svg> node, plus we will be able to easily access the _data array itself in the *enter/update/exit logic,* since the _data array is now the value 'd' in those callbacks as shown here:

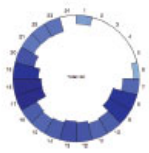
```
001: // Enter, Update, Exit on bars
002: var bars = svg.selectAll(".bar")
003:     .data(function(d, i) { return d; /* d === _data */ });
```

The remainder of the code in the 03.reusable-chart-with-transition-support.html is a simple expansion on the bar chart plugin code shown in this chapter, only extended with animation (d3.transition) support.

Summary

This chapter more fully fleshes out how to use the reusable API to construct reusable charts. In the next chapter we move onto discussing how to use test suites to fully test the plugins, modules, and components that we create.

5. Unit Testing / Test Suites



IN THIS CHAPTER

- Build a test suite using the Jasmine BDD framework
- Test the bar chart plugin developed in previous chapters

Building Testable Code

Implementing the reusable API will help with building testable code. We will assume you already know what Test-Driven Development (TDD) is, or at least what is a *unit test*. TDD forces you to write modular units, that are totally decoupled from other modules, where the logic is usually hidden behind a clean API. Using a test suite framework or a simple *assert* function, you write down the *contract (*the way to use your module from the public API), in a way that is meaningful and readable for a user. Here is one unit test example:

```
001: it('should add a number to another', function() {
002:     expect(addFunctionToTest(2, 3)).toBe(5);
003: });
```

TDD is more than just a way to document your code. You can be more aggressive in your refactoring when you know that a test will fail, since you will be informed if you break the contract other modules potentially rely upon. A good test suite, though, is the best documentation you can have. If you've never read the D3 test suite, now is a good time to do so.

In this chapter we will write a simple test suite for the reusable chart plugin API we discussed in the previous chapter (Chapter 4) to show some of its key features in action. We will use the Jasmine BDD framework, but it should be easy to port this test suite to any other TDD/BDD framework. Keep in mind that this is not a guide about how to write a good test suite, but instead will illustrate and document topics we have discussed so far in this book.

A Test Suite for our Bar Chart Plugin

NOTE Source code is available in code/chapter05

A test suite usually starts with a description and the initialization of some commonly used variables, as well as with an HTML fixture that is used as a sandbox.

```
001: describe('Reusable Bar Chart Test Suite', function() {
002:   var barChart, dataset, fixture;
003:
004:   beforeEach(function () {
005:     dataset = [10, 20, 30, 40];
006:     barChart = d3.edge.barChart();
007:     fixture = d3.select('body').append('div').classed('test-container', true);
008:   });
009:
010:   afterEach(function () {
011:     fixture.remove();
012:   });
013: });
```

Let's first test some basic usage. We only need one line of code (and two more for the test harness itself). We bind some data to the DOM fixture and call our chart function. Note that the d3.datum() method is like d3.selection.data(), but doesn't compute a *data join*, which we don't need in this case:

```
001: it('should render a chart with minimal requirements', function() {
002:   fixture.datum(dataset).call(barChart);
003:   expect(fixture.select('.chart')).toBeDefined(1);
004: });
```

Setters are also used as getters when no arguments are provided; this is where we test for that dual behavior:

```
001: it('should provide getters and setters', function() {
002:   var defaultWidth = barChart.width();
003:   var defaultEase = barChart.ease();
004:
005:   barChart.width(1234).ease('linear');
006:
007:   var newWidth = barChart.width();
008:   var newEase = barChart.ease();
009:
010:   expect(defaultWidth).not.toBe(1234);
```

```
    expect(defaultEase).not.toBe('linear');
    expect(newWidth).toBe(1234);
    expect(newEase).toBe('linear');
  });
```

Here we just want to demonstrate that some functions are not accessible from the API:

```
it('should scope some private and some public fields and methods', function() {
  expect(barChart.className).toBeUndefined();
  expect(barChart.ease).toBeDefined();
  expect(typeof barChart.ease).toBe('function');
});
```

We update an attribute on the chart by using a setter and calling the chart function again. Here, we only test if the new attribute has been set inside the module, and not if the new width was actually applied to the chart. This is slightly more involved and is sometimes better tested with browser automation and automated screenshot comparison tools:

```
it('should update a chart with new attributes', function() {
  barChart.width(10000);
  fixture.datum(dataset)
    .call(barChart);

  barChart.width(20000);
  fixture.call(barChart);

  expect(barChart.width()).toBe(20000);
});
```

Updating the data is slightly different. We have to bind the new data to the DOM *before* calling our chart function. It's not good practice for a unit test to have too much knowledge about the internal working of a module, but here we illustrate that the data, once bound to the DOM, is available under the special `__data__` field of the DOM object. Another trick to mention here is the use of `[0][0]` to access the first member of a selection, illustrating that this selection is an array of DOM elements:

```
it('should update a chart with new data', function() {
  fixture.datum(dataset)
    .call(barChart);

  var firstBarNodeData1 = fixture.selectAll('.bar')[0][0].__data__;

  var dataset2 = [1000];
  fixture.datum(dataset2)
    .call(barChart);

  var firstBarNodeData2 = fixture.selectAll('.bar')[0][0].__data__;

  expect(firstBarNodeData1).toBe(dataset[0]);
  expect(firstBarNodeData2).toBe(dataset2[0]);
});
```

Now let's render two charts in two `<div>`s. We first verify if they are both there with a different configuration:

```
it('should render two charts with distinct configuration', function() {
  fixture.append('div')
    .datum(dataset)
    .call(barChart);

  var dataset2 = [400, 300, 200, 100];
  var barChart2 = d3.edge.barChart().ease('linear');

  fixture.append('div')
    .datum(dataset2)
    .call(barChart2);

  var charts = fixture.selectAll('.chart');

  expect(charts[0].length).toBe(2);
  expect(barChart2.ease()).not.toBe(barChart.ease());
});
```

With this in mind, let's look at a less obvious usage example. A reusable chart can be used as a component in another chart, as is the case with the `d3.axis` component. Here we add a chart inside another chart:

```
it('can be composed with another one', function() {
  fixture.datum(dataset)
    .call(barChart);

  var barChart2 = d3.edge.barChart();

  fixture.selectAll('.chart')
    .datum(dataset)
    .call(barChart2);

  var charts = fixture.selectAll('.chart');

  expect(charts[0].length).toBe(2);
  expect(charts[0][1].parentElement).toBe(charts[0][0]);
});
```

Another interesting usage we didn't describe earlier is the use of `_selection.each`, which is used to loop through the selection. Its task is to build a chart for each element of the selection set that was passed to the chart module. Next, we will pass an array of arrays. The `.each` loop will add a child `<div>` to the container for each of the three sub-arrays:

```
it('should render a chart for each data series', function() {
  var dataset = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]];

```

```
004:   fixture.selectAll('div.container')
005:     .data(dataset)
006:     .enter().append('div')
007:     .classed('container', true)
008:     .datum(function(d, i){return d;})
009:     .call(barChart);
010:
011:   var charts = fixture.selectAll('.chart');
012:
013:   expect(charts[0].length).toBe(dataset.length);
014:   expect(charts[0][0].__data__).toBe(dataset[0]);
015:   expect(charts[0][1].__data__).toBe(dataset[1]);
016:   expect(charts[0][2].__data__).toBe(dataset[2]);
017: });
```

We call to the chart only once, but internally it is built three times, each time with a different dataset. This gives you small multiples for free!

Testing Events: Using Spies

Callbacks on events are a bit harder to test. We use a *spy* for this task and watch for it, since it is called with the expected arguments when the event is triggered. The tricky part is to trigger the event. D3.js comes to the rescue, since the custom events we bound with d3.dispatch are available as properties of the DOM object. Note how in this specific example they exist under the name __onmouseover():

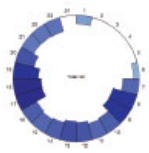
```
001: it('should trigger a callback on events', function() {
002:   fixture.datum(dataset)
003:     .call(barChart);
004:
005:   var callback = jasmine.createSpy("filterCallback");
006:   barChart.on('customHover', callback);
007:
008:   var bars = fixture.selectAll('.bar');
009:   bars[0][0].__onmouseover();
010:   var callBackArguments = callback.argsForCall[0][0];
011:
012:   expect(callback).toHaveBeenCalled();
013:   expect(callBackArguments).toBe(dataset[0]);
014: });
```

Voilà! We have illustrated some features provided by the reusable API using a unit test suite. It's always a good idea to add a unit test when you share a plugin, and it is mandatory when working on the D3.js core.

Summary

If you made it this far into this chapter, you are probably geeky enough to already be convinced by the virtues of TDD/BDD. So let's go on to the next chapter and build a real-world application using the reusable API.

6. Plugin Example

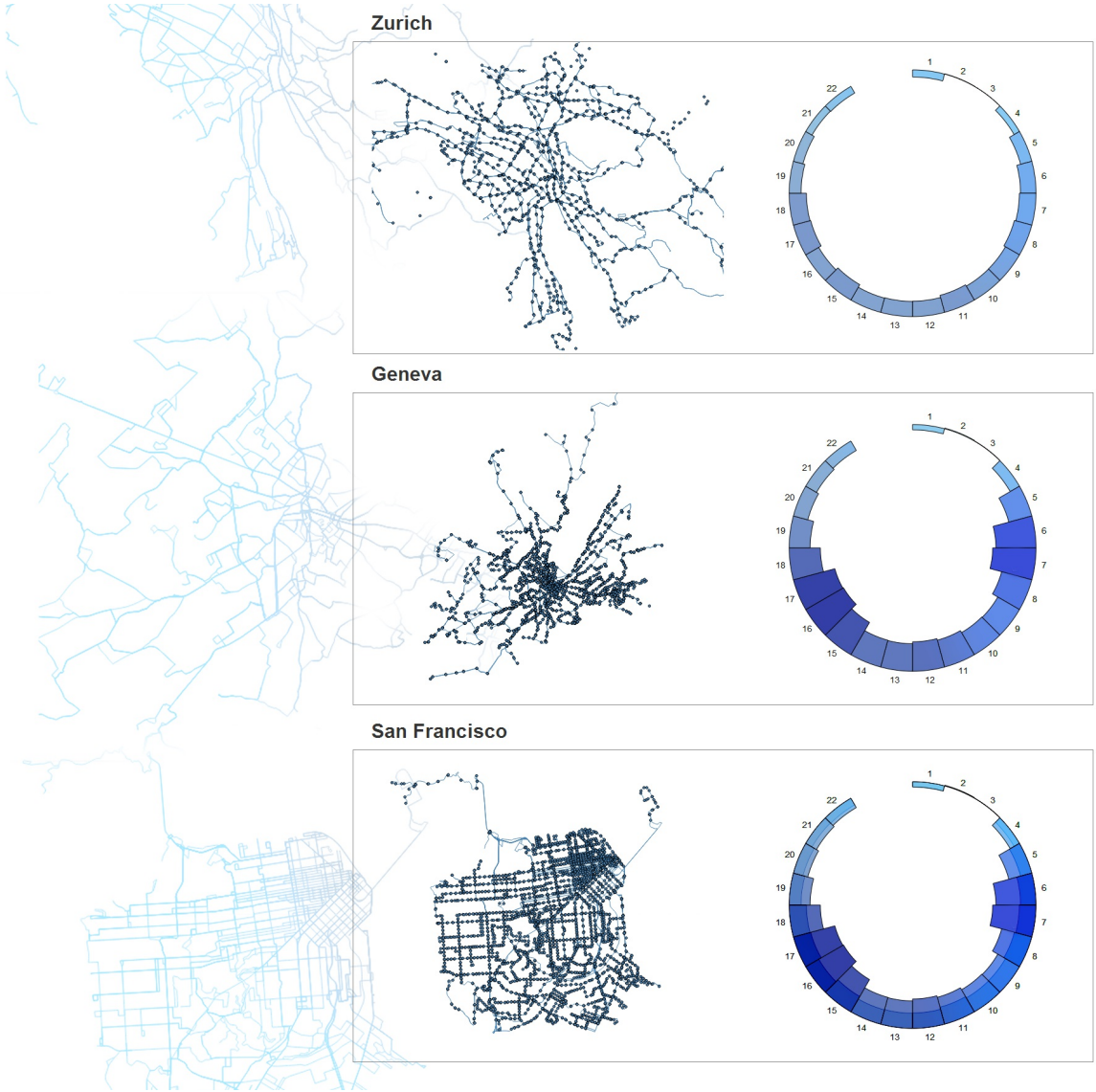


IN THIS CHAPTER

- Introduce a set of real world data
- Implement a visualization of real world data with the reusable API

We have now covered the basics of how to implement the reusable API, seeing first hand the benefits of using it. It is now time to look at a real world example: in this case, a dataset provided by a challenge from Urban Prototyping. The data will be used to provide valuable insight into the transportation challenges of three major cities around the world: Geneva, Zurich, and San Francisco. The organizers of the challenge have provided geographic data and key metrics for transit routes throughout the cities.

This data set provides several dimensions that we wish to explore. We have geographic data, time data, and several metrics provided at each transit stop. A data set like this lends itself to multiple, linked visualizations. We've chosen to use a map to illustrate the geographic dimension, and a radial histogram to simultaneously visualize the time dimension and stop metrics. An impression of the final product can be seen in the figure below.



Why A Reusable Approach?

This challenge provides a perfect opportunity to implement the reusable API, since it poses several challenges and issues that a reusable API can address.

The most obvious issue is the fact that we have to recreate our visualization three times, once for each city. If we were to duplicate the code to produce each visualization three times, our code would quickly become unmanageable and bloated. For example, if we wanted to make a change to the map for each city, we would have to implement this change three times. With the reusable approach, we can create a reusable map module and make an amendment to the module, and the change will be propagated through every visualization.

Next, we have fairly large data sets (> 50 megabytes) that can be cumbersome for the browser to deal with efficiently. These data sets take a long time to load, require some post-load cleaning, and filtering them natively can be tedious. There are several libraries that make this task easier. We can define a reusable module to deal with these tasks for each of our three data sets, and integrate these libraries within the module to abstract this process for our custom API.

Finally, to tie everything together, we need some custom events that allow us to respond to data loading events, hovering events, zooming events and brushing events. By implementing these into our reusable modules, we are able to simplify this process across all three visualizations and create a common interface within every visualization.

Before any of this can begin, however, we first need to define our requirements.

API Requirements

The first step in developing a reusable API is to define the requirements it should meet, along with any expectations you may have.

We want an application that can efficiently visualize a large, multi-variate data set, linking key dimensions for an all encompassing view of the data. In addition, it should be highly adaptable, keeping a simple API to encapsulate an implementation smart enough to handle very different configurations. This will allow us to simply modify the input data, and have the visualization update accordingly. From this, we have an API that must:

1. Efficiently load and clean large (> 50 megabyte) data sets.
2. Summarize the data using third-party libraries.
3. Produce a map of the geographic data.
4. Produce a radial histogram of the time, and stop metric data.
5. Link the map and radial histogram to the transit stop data.
6. Be straightforward to adapt to new input data.

Now that we understand what is required of our API, we can begin the process of defining it.

Breaking The Application Into Modules

Now that we have specified the requirements of our reusable API we can begin the next step of breaking it into specific modules. With a complex visualization like this, it's often best to start simple and take some time to think about how to break the visualization into smaller pieces. For our example we know we need to:

1. Load the data into the browser and apply some data cleaning.
2. Generate a map of each city to visualize the geographic dimension.
3. Generate a graph to visualize key metrics about various stops.
4. Link the two visualizations together to allow dynamic data exploration.

Accomplishing all of this without using the reusable pattern would be a monumental task, but fortunately, the reusable pattern dramatically simplifies this process. Let's take a look at how to get started.

Cleaning The Data

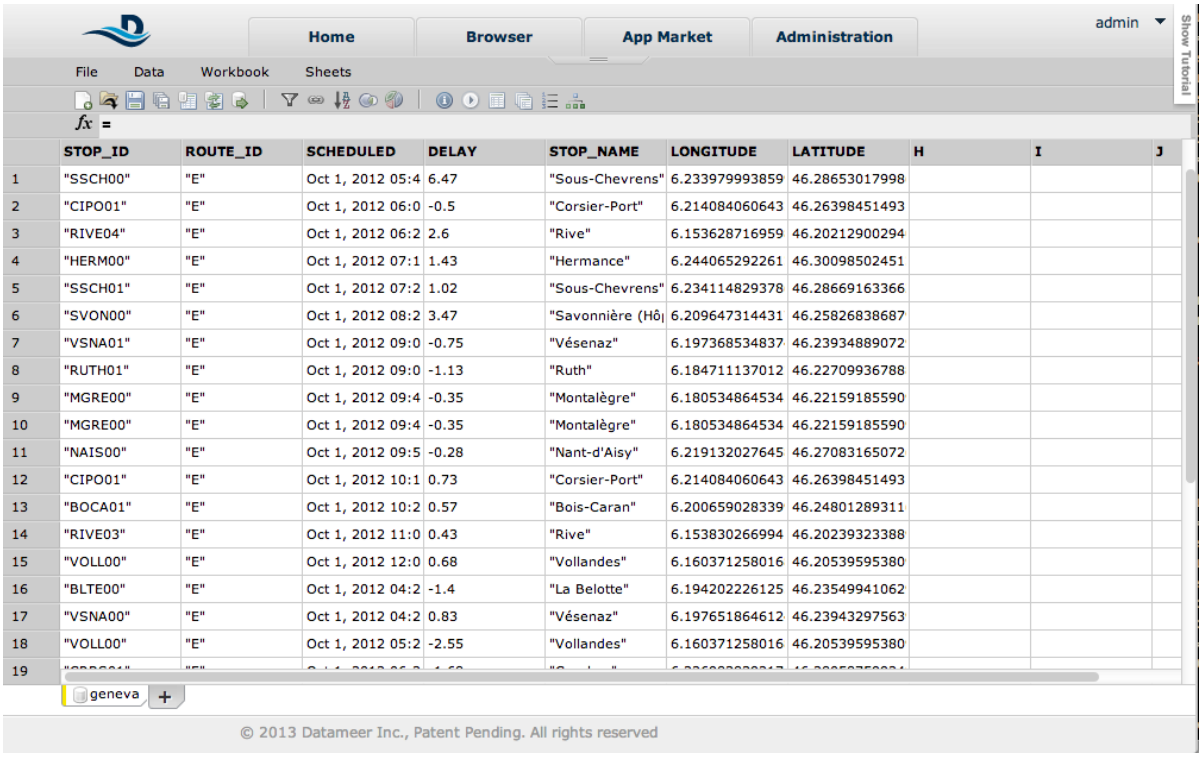
Visualization is more than rendering graphics. It often starts with a human having a question. The data is a mediary between the human and the object of the study. Visualization will help to extract knowledge from this data. But first, the data should be good at representing the object to study. Collecting, storing, cleaning transforming, and computing statistics are the important steps for preparing a good visualization. Most of the D3.js examples you can find on the web come with clean data, or even generated data. But, as with most data you will encounter in real life, the dataset for our example required some cleaning before it could be used.

NOTE The cleaned data can be found on the github repository that also contains the source code for this book: all data sources are located in the data/ directory tree.

To visualize a dataset using D3.js, go through the following steps:

- Collecting
- Loading a data file or connecting to a data source (i.e., using d3.json)
- Cleaning (i.e., joining, denullifying, standardizing, formatting, using simple Array an Object methods or specialized tools)
- Converting the data to the right structure for a D3.js layout (i.e., CSV to matrix, flat JSON to hierarchical using d3.nest)
- Mapping from data to layout space (i.e., using d3.layout.stack to inject some layout information to the dataset)
- Binding data to the DOM object (i.e., using d3.selectAll().data())
- Mapping from layout to pixel space (i.e., using d3.scale to map from layout space to pixel space)
- Render graphical attributes in the DOM using this prepared data (i.e., using d3.attr)

As previously mentioned, the Urban Data Challenge dataset needed some cleaning and processing, which is most often the case in real-world scenarios. For example, the San Francisco dataset had the real-time and scheduled bus arrival time in two different files, unlike the Geneva and the Zurich dataset. So, to compute the bus delay, the files had to be joined, even if no join index was readily available. The Geneva dataset also needed a longitude and a latitude column, which were only available in the geojson files. Time filtering and removing null values are typical operations for cleaning data that we had to perform. Specialized tools like OpenRefine, or generic tools like LibreOffice Calc, are often used for this. The file was too big for loading into Calc or Excel, so we used Datameer, a spreadsheet-based analytics tool for BigData. Although Datameer was developed specifically for huge datasets (like petabytes of data from Hadoop clusters), their filtering, joining, grouping and more than 200 analytics functions make it a good candidate for the task (*full disclosure: one of the authors is a Datameer developer)*.

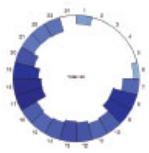


D3.js can load dirty data and clean it on the fly. But for a dataset totaling 180 MBs over 3 files, reducing it to 60 MBs of clean data was still heavy loading with Ajax. We could have reduced the file size more by using just a subset. But once gzipped, the data size was acceptable for this demo application. And it is a good illustration of a typical bottleneck in the visualization process, especially for online applications, which has to be solved with tools outside of the D3.js ecosystem.

Summary

Now that the data sets for each city have been cleaned up and contain similar fields that allow for comparison, we can implement the reusable pattern to construct a useful visualization that allows for exploration of the data, which we will do in the next chapter.

7. Data Manager API



IN THIS CHAPTER

- Introduce the data manager
- Build a reusable component not bound to any visual element
- Discuss custom events

The Data Manager

NOTE The source code is available in the code/Chapter07/.

For our example, let's introduce the concept of a data manager. The data manager will be responsible for loading the data into the browser, applying any post-loading cleaning of the data that may be required, filtering the data based upon user interaction, and letting us know when the data is ready. The data manager also serves to demonstrate that the reusable concept can be applied to more than just graphs. It can also be used for any number of tasks that will be repeated throughout the visualization.

For our application, all of our reusable modules are namespaced under:

```
001: var d3Edge = {};
```

This will ensure that none of the modules that are introduced will conflict with other linked libraries.

To start, our first task is to setup the data manager module:

```
001: d3Edge.dataManager = function module() {
002:     var exports = {},
003:         dispatch = d3.dispatch('geoReady','dataReady','dataLoading'),
004:         data;
005:     d3.rebind(exports, dispatch, 'on');
006:     return exports;
007: };
```

Here, we have created a function under the d3Edge namespace called: dataManager. This function will return an object called exports that will contain various methods defined later. To instantiate this module in our application, simply assign the function to a new variable:

```
001: var zurichDataManager = d3Edge.dataManager();
```

In addition, we added three custom events to this module that will give us some indication of when various events in the data loading process are occurring. To accomplish this, we are using d3.dispatch to create three events: *geoReady, dataReady*, and *dataLoading*. These events are bound to the 'on' method of the exports object using d3.rebind. We can fire these events anywhere within our module at a time of our choosing by calling the dispatch function together with the name of the event. For instance, for the *dataReady* event*:*

```
001: dispatch.dataReady();
```

We can then listen for the event on the instantiated module like so:

```
001: zurichDataManager.on('dataReady', function(data) {
002:     // Do Something With The Data Here.
003: });
```

This sets the basic framework up for the data manager module. Now we need to create methods on our exports object that can be used by our instantiated zurichDataManager. Our first method will be to load the CSV file from the server, apply a cleaning function, and fire an event to indicate that the data is ready. In addition, we will create a simple getter method that will allow us to access the cleaned data at anytime.

```
001: exports.loadCsvData = function(_file, _cleaningFunc) {
002:     // Create the csv request using d3.csv.
003:     var loadCsv = d3.csv(_file);
004:     // On the progress event, dispatch the custom dataLoading event.
005:     loadCsv.on('progress', function() {
006:         dispatch.dataLoading(d3.event.loaded);
007:     });
008:     // Issue a HTTP get for the csv file, and work with the data.
009:     loadCsv.get(function (_err, _response) {
010:         // Apply the cleaning function supplied in the _cleaningFunc parameter.
011:         _response.forEach(function (d) {
```

```
            _cleaningFunc(d);
        });
        // Assign the cleaned response to our data variable.
        data =_response;
        // Dispatch our custom dataReady event passing in the cleaned data.
        dispatch.dataReady(_response);
    });
};
//Create a method to access the cleaned data.
exports.getCleanedData = function () {
    return data;
};
```

We first create our method loadCsvData by defining it on the exports object. This method accepts two parameters: _file and _cleaningFunc. This allows us to easily pass a different file path and cleaning function for each instantiation of the module. This is important as our data sets do not always contain the same column headers, and the cleaning function must be modified to fit each data set. Once the cleaning function is asynchronously applied to the response, we assign the result to our variable data. Our method getCleanData will simply return the variable data when invoked, thus providing easy access to the cleaned data for use or inspection.

For the three data sets we instantiate and load the data like so:

```
//Instantiate our data manager module for each city.
var sanFranciscoDataManager = d3Edge.dataManager(),
    zurichDataManager = d3Edge.dataManager(),
    genevaDataManager = d3Edge.dataManager();

//Load our Zurich data, and supply the cleaning function.
zurichDataManager.loadCsvData('../__data/zurich/zurich_delay.csv', function(d){
  var timeFormat = d3.time.format('%Y-%m-%d %H:%M:%S %p');
  d.DELAY = +d.DELAY_MIN;
  delete d.DELAY_MIN;
  d.SCHEDULED = timeFormat.parse(d.SCHEDULED);
  d.LATITUDE = +d.LATITUDE;
  d.LONGITUDE = +d.LONGITUDE;
  d.LOCATION = [d.LONGITUDE, d.LATITUDE];
});

//Load our Geneva data, and supply the cleaning function.
genevaDataManager.loadCsvData('../__data/geneva/geneva_delay_coord.csv', function(d){
  var timeFormat = d3.time.format('%Y-%m-%d %H:%M:%S %p');
  d.DELAY = +d.DELAY;
  d.SCHEDULED = timeFormat.parse(d.SCHEDULED);
  d.LATITUDE = +d.LATITUDE;
  d.LONGITUDE = +d.LONGITUDE;
  d.LOCATION = [d.LONGITUDE, d.LATITUDE];
});

//Load our San Francisco data, and supply the cleaning function.
sanFranciscoDataManager.loadCsvData('../__data/san_francisco/san_francisco_delay.csv', function(d){
  var timeFormat = d3.time.format('%Y-%m-%d %H:%M:%S %p');
  d.DELAY = +d.DELAY_MIN;
  delete d.DELAY_MIN;
  d.SCHEDULED = timeFormat.parse(d.SCHEDULED);
  d.LATITUDE = +d.LATITUDE;
  d.LONGITUDE = +d.LONGITUDE;
  d.LOCATION = [d.LONGITUDE, d.LATITUDE];
});
```

In the code above we first instantiate our reusable data manager for each city. We then load and clean the data for each city by passing in the URL for the CSV, along with the cleaning function for the respective cities. For each data file the headers are slightly different, thus complicating the cleaning operation. Luckily, our API is flexible enough to accommodate this by allowing us to pass in separate cleaning functions for each city. If we call our getter method, getCleanedData, we should see correctly formatted data for each city.

```
// Calling the getCleanedData Method to inspect the first element of the data.
zurich.getCleanedData()[0];
// Example output showing parsed numbers and new LOCATION data field.
{
    "ROUTE_ID": "305",
    "ROUTE_NAME": "2",
    "STOP_ID": "2251",
    "STOP_NAME_SHORT": "DEP4",
    "STOP_NAME": "Zürich, Depot 4 Elisabethenstr",
    "SCHEDULED": "2012-10-01T11:00:00.000Z",
    "LONGITUDE": 8.521632222222221,
    "LATITUDE": 47.37333861111111,
    "DELAY": 0.18,
    "LOCATION": [ 8.521632222222221, 47.37333861111111 ]
}
```

We now have our stop data loaded into the browser, cleaned, and easily accessible. The next step is to do the same for the geometric data. In our case, the geometric data comes in the geoJSON format for both the routes, and the stops. Therefore, we are able to use the same method to load these data files and simply pass in a callback that can be tailored to fit the needs of each file once we receive the response. To do this, we are going to define another method on the exports object called loadgeoJSON. This method will accept two arguments: _file and _callback. The _file argument will simply be the path to the data on our server. The _callback argument will be the custom callback that will executed asynchronously for each data manager once the data file is finished loading. We construct this method like so:

```
// Create a method to load the geoJSON file, and execute a custom callback on response.
exports.loadgeoJSON = function(_file, _callback) {
    // Load json file using d3.json.
    d3.json(_file, function (_err, _data) {
        // Execute the callback, passing in the data.
        _callback(_data);
    });
};
```

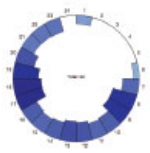
We can invoke this method by simply calling the `loadgeoJSON` method on our instantiated data managers for each city.

```
001: // Load the routes data and pass in the callback to be executed once the data loads.
002: zurichDataManager.loadgeoJSON('./data/zurich/routes_topo.json', function (data) {
003:     // Do something with the data via a callback.
004: });
005: // Load the stops data and pass in the callback to be executed once the data loads.
006: zurichDataManager.loadgeoJSON('./data/zurich/stops_geo.json', function (data) {
007:     // Do something with the data via a callback.
008: });
```

Summary

We now have our data in the browser, we have cleaned it, and we have created custom events to indicate when this whole process is complete! We can now begin the process of bringing this data to life, which we will do in the next chapter.

8. Map API



IN THIS CHAPTER

- Visualize our data in a map
- Produce a map for all three cities
- Plot the routes and stops for each city

As outlined in our API's requirements, we want to visualize our data in a map, and in a radial histogram that aggregates a metric of our choosing. We will first take a look at producing the API for the map module. For our reusable map module, we obviously want to use it to produce a map for each individual city, and plot the routes and stops for each city.

To accomplish this, the module must be able to accept varying data sets, be centered on the city's coordinates, scaled to appropriately display the data points, have zoom capability, and be sized to customize our needs. In addition, later on we want to add some brushing, which will allow us to link the two graphics together for each city.

Drawing Routes

NOTE Source code and data files are available in the code/Chapter08/DrawingRoutes/ directory tree.

To begin with, we will define our map module:

```
001: d3Edge.map = function module() {
002:     // Create our custom events, and variables.
003:     var dispatch = d3.dispatch('hover', 'drawEnd', 'brushing'),
004:         projection,
005:         path,
006:         t,
007:         s,
008:         svg,
009:         center,
010:         scale,
011:         size,
012:         brush;
013:     // Create and exports function that can be invoked on a selection.
014:     function exports(_selection) {
015:     }
016:     // Bind our custom events to the 'on' method of our function.
017:     d3.rebind(exports, dispatch, 'on');
018:     return exports;
019: };
```

You may have noticed that in this module, we return a function rather than an object, like we did in our data manager module.

In the data manager module, exports was defined as an object:

```
001: var exports = {},
```

In all of the earlier examples in this book, exports is defined as a function:

```
001: function exports(....) {
```

We take the approach here (i.e. returning a function and not an object), because this module will be *invoked* on a D3.js selection. This will allow us to perform standard D3.js DOM manipulation and data binding, thus actually creating our visualization.

Now that we have our module's skeleton code defined, we can begin implementing our methods. First off, as we mentioned above, we need to be able to center, scale, and size each map. To set these parameters, we will create a method for each map that will serve as both a getter and a setter. Each method accepts an argument. If the argument is supplied, the parameter is updated with the value of the argument (*setter*). If the argument is not supplied, the current value of the parameter is returned (*getter*):

```
001: // Create a center method to serve as both a getter, and a setter.
002: exports.center = function(_x) {
003:     if (!arguments.length)
004:         return center;
005:     center = _x;
006:     return this;
007: };
```

```
// Create a scale method to serve as both a getter, and a setter.
exports.scale = function(_x) {
    if (!arguments.length)
        return scale;
    scale = _x;
    return this;
};
// Create a size method to serve as both a getter and setter.
exports.size = function(_x) {
    if (!arguments.length)
        return size;
    size = _x;
    return this;
};
```

By returning this when updating a parameter, we allow ourselves to easily chain these methods together for succinct code. This makes instantiating each instance of our map module very simple. For example, a map for Zurich could be instantiated, centered, scaled, and sized like so:

```
zurichMap = d3Edge.map()
.center([8.5390, 47.3687])
.scale(900000)
.size([width, height]);
```

This pattern should be very familiar to you since it is one D3.js uses very often. However, instantiating this won't produce much since we haven't defined any mapping yet! Our next step is to develop our exports function so that we can actually produce a visualization. Since we want this module to be used for generating both the routes and the stops, our export function will simply be used to access the svg element that calls our module, and to define our projection and path generating functions using d3.geo.mercator() and d3.geo.path():

```
// Create and exports function that can be invoked on a selection.
function exports(_selection) {
    // Set svg equal to the selection that invokes this module.
    svg = svg || _selection;
    // Bind an empty datum to the selection. Useful later for zooming.
    svg.datum([]);
    // Set the projection up using our scale, center, and size parameters.
    projection = projection || d3.geo.mercator()
    .scale(scale)
    .center(center)
    .translate([size[0]/2, size[1]/2]);
    // Set the path up using our projection defined above.
    path = path || d3.geo.path()
    .projection(projection);
}
```

The exports function above is simply grabbing onto the svg element that calls our module, and setting up our projections. If the projections are already defined, we will us them. If not, we have supplied defaults.

We can then call our module from a D3.js selection like so:

```
var width = 570,
    height = 500;

var zurichMap = d3Edge.map()
.center([8.5390, 47.3687])
.scale(900000)
.size([width, height]);

d3.select('#zurich_map')
.append('svg') .attr('width', width)
.attr('height', height)
.call(zurichMap);
```

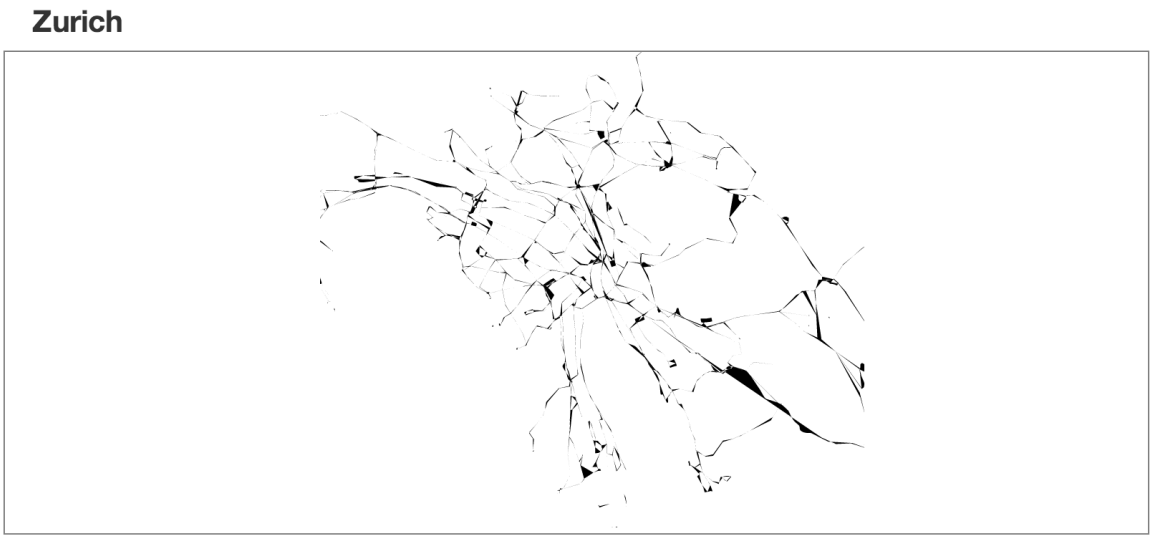
At this point, our exports function has granted us access to the svg element that called our module and assigned it to a local variable, thus allowing us to apply standard D3.js methods to produce a visualization. We can now use the local variable, svg, throughout our module to append both our routes and our stops. Let us define a method that will display our routes:

```
// Create a drawRoutes method that can be invoked to create routes for each city.
exports.drawRoutes = function(_data) {
    svg.append('path')
    .attr('class', 'route')
    .datum(topojson.object(_data, _data.objects.routes))
    .attr('d', function(d, i) {
        return path(d, i);
    });
    // Dispatch our routesEnd event so we know with the routes visualization is complete.
    dispatch.routesEnd();
};
```

This method accepts a single argument--the geographic data to be mapped. It then appends a path to our local svg variable, using the path generator as defined in our exports function. If we invoke this method, we will finally start to see something on our screen! It is at this point we can combine our data manager module with our map module. Since the drawRoutes method needs geographic data as it's only argument, we can invoke it as the callback of our loadGeoJson method on our data manager module:

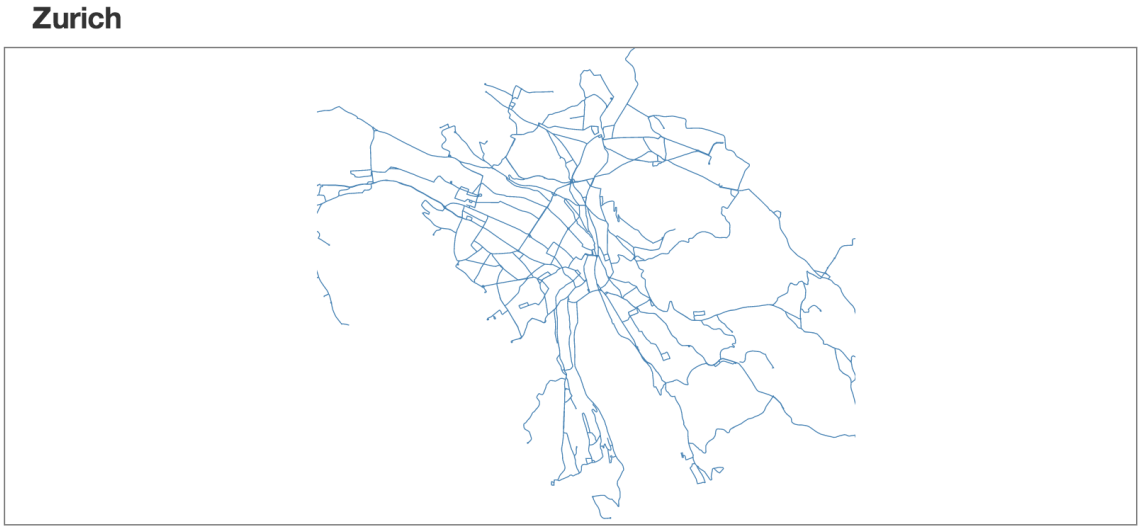
```
// Load the routes data and pass our drawRoutes method as the callback to be executed upon data load.
zurichDataManager.loadGeoJson('./data/zurich/routes_topo.json', zurichMap.drawRoutes);
```

Executing this code will produce the transit routes for Zurich as shown in the image below.



Applying a little bit of CSS (style sheets) produces something a little more appealing.

```
.route {
    stroke: #4682b4;
    fill: none;
}
```



Drawing Stops

NOTE The source code and data files are available in the code/Chapter08/DrawingStops/ directory tree.

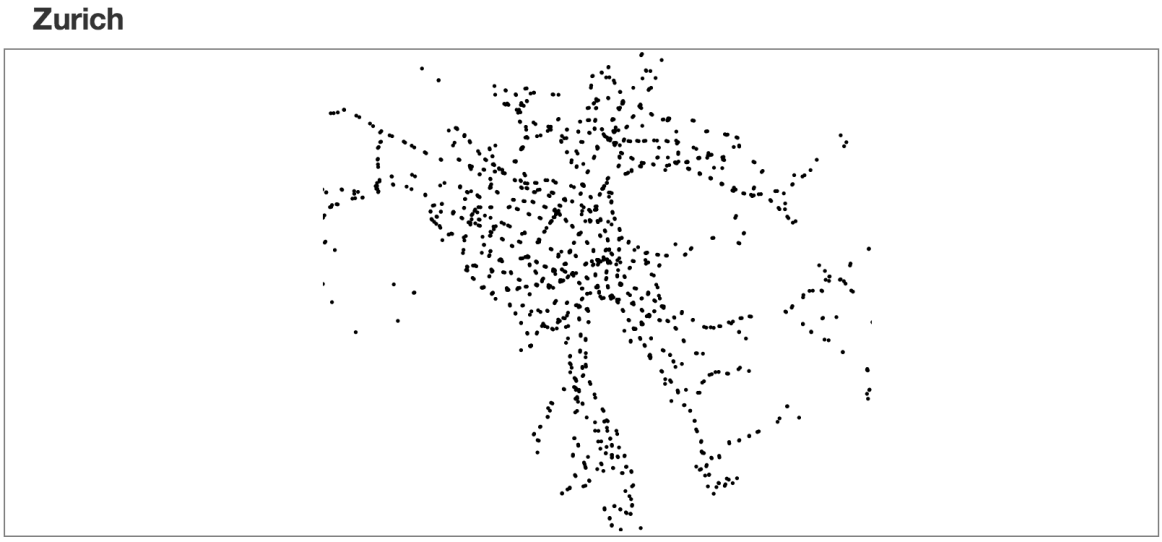
We can create a similar method to visualize the stops. In this method, however, we want to add a custom event to allow us to interact with individual stops on mouseover:

```
// Create a drawStops method that can be invoked to create stops for each city.
exports.drawStops = function(_data) {
    svg.selectAll('.stop')
    .data(_data.features)
    .enter()
        .append('circle')
        .attr('cx', function (d) { return projection(d.geometry.coordinates)[0]; })
        .attr('cy', function (d) { return projection(d.geometry.coordinates)[1]; })
        .attr('r', 2)
        .attr('class', 'stop')
        .on('mouseover', dispatch.hover);
    // Dispatch our stopsEnd event so we know with the stops visualization is complete.
    dispatch.stopsEnd();
};
```

Just like the `drawRoutes` method, this method accepts a single argument, the data to be mapped. For each data point, it will plot a point at using the projection we defined in our exports function. In addition, this method we have added a custom event that will be fired when we mouseover a circle. The definition of how to handle this event can

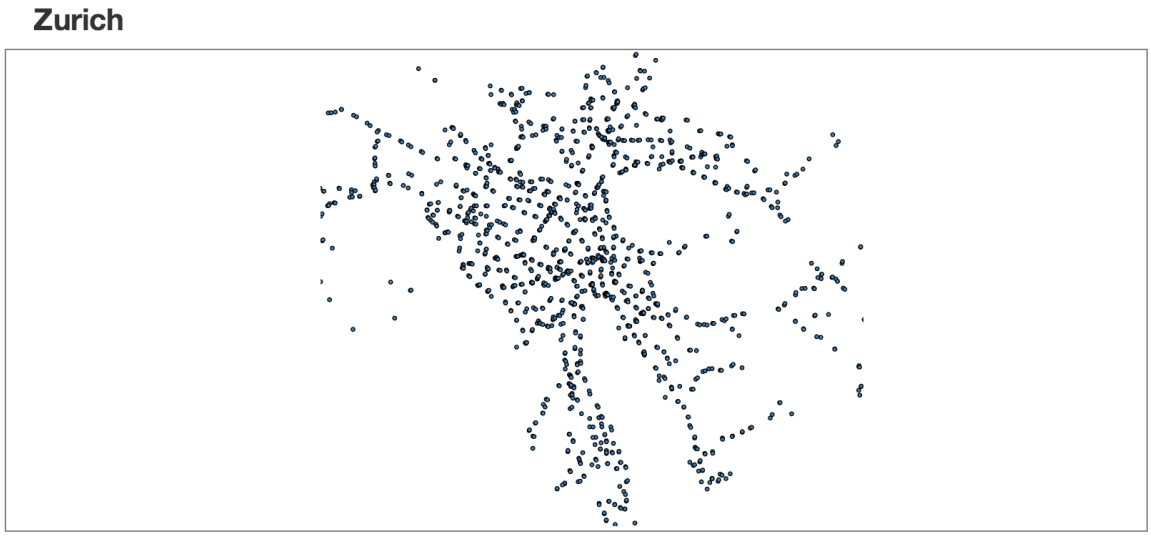
then be defined when we instantiate our mapping module for each instance of the chart.

Executing this code will produce the transit stops for Zurich as shown here:



Again, we can apply a little CSS to make things a little more appealing.

```
.stop {
    fill: #4682b4;
    stroke: #000;
    cursor: pointer;
}
```



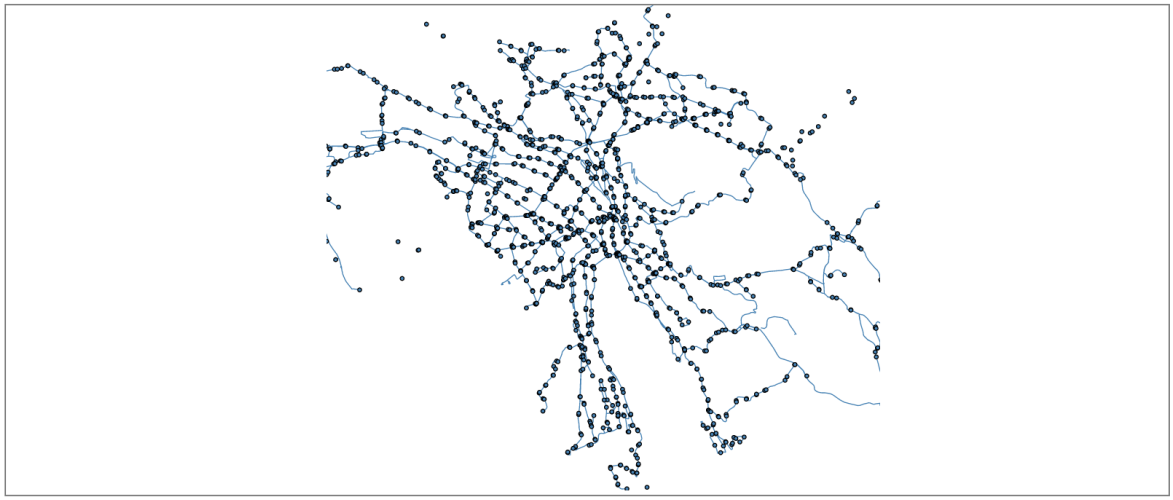
Draw Route and Stops

NOTE The source code and data files are available in the code/Chapter08/DrawRoutesAndStops/ directory tree.

Finally, we can combine these two methods to create a complete map. We will use our custom *routesEnd* event to *invoke* our drawStops method after the routes have been rendered. This ensures that the stops appear on top of the routes:

```
// Load the routes data and pass our drawRoutes method as the callback to be executed once the data loads.
zurichDataManager.loadGeoJson('./data/zurich/routes_topo.json', zurichMap.drawRoutes);
// After the routes have been rendered, render the stops.
zurichMap.on('routesEnd', function () {
    // Load the stops data and pass our drawStops method as the callback to be executed once the data loads.
    zurichDataManager.loadGeoJson('./data/zurich/stops_geo.json', zurichMap.drawStops);
});
```

Zurich



Combine The Maps

NOTE Source code and data files are available in the code/Chapter08/CombineTheMaps/ directory tree.

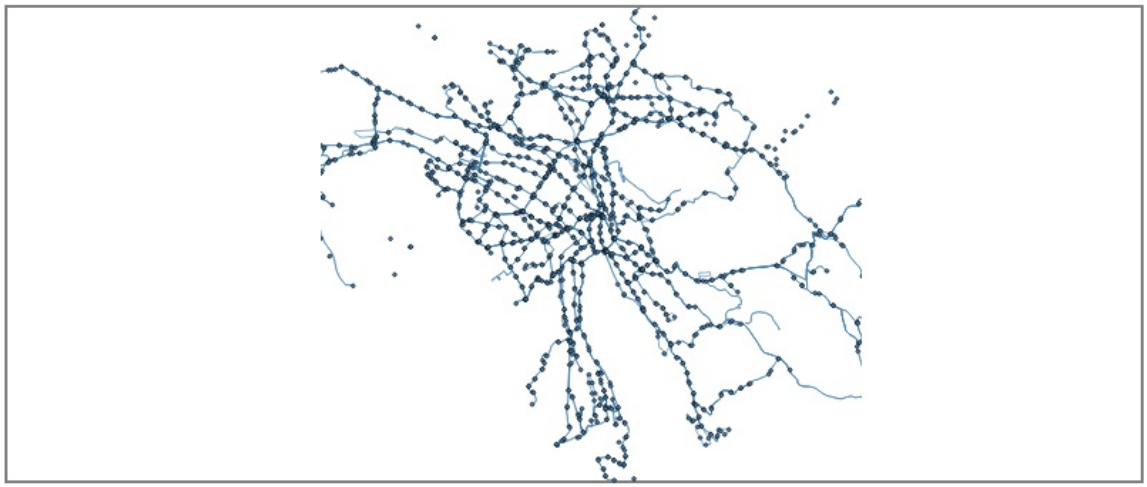
Now that our map module is producing something worth looking at, let's leverage the power of the module and create a map for each city. All we need to do is instantiate a data manager module and a map module for each city, and call the methods we outlined above:

```
// Define our width and height for our visualizations.
var width = 570,
height = 500;
// Instantiate our data manager module for each city.
var sanFranciscoDataManager = d3Edge.dataManager(),
    zurichDataManager = d3Edge.dataManager(),
    genevaDataManager = d3Edge.dataManager();
// Instantiate our map module for Zurich.
var zurichMap = d3Edge.map()
    .center([8.5390, 47.3687])
    .scale(900000)
    .size([width, height]);
// Instantiate our map module for Geneva.
var genevaMap = d3Edge.map()
    .center([6.14, 46.20])
    .scale(900000)
    .size([width, height]);
// Instantiate our map module for San Francisco.
var sanFranciscoMap = d3Edge.map()
    .center([-122.4376, 37.77])
    .scale(900000)
    .size([width, height]);
// Bind our modules to the DOM.
d3.select('#zurich_map')
.append('svg')
.attr('width', width)
.attr('height', height)
.call(zurichMap);

d3.select('#geneva_map')
.append('svg')
.attr('width', width)
.attr('height', height)
.call(genevaMap);

d3.select('#san_francisco_map')
.append('svg')
.attr('width', width)
.attr('height', height)
.call(sanFranciscoMap);

// Load the routes data and pass our drawRoutes method as the callback to be executed once the data loads.
zurichDataManager.loadGeoJson('./data/zurich/routes_topo.json', zurichMap.drawRoutes);
// After the routes have been drawn, draw the stops.
zurichMap.on('routesEnd', function () {
    // Load the stops data and pass our drawStops method as the callback to be executed once the data loads.
    zurichDataManager.loadGeoJson('./data/zurich/stops_geo.json', zurichMap.drawStops);
});

// Load the routes data and pass our drawRoutes method as the callback to be executed once the data loads.
genevaDataManager.loadGeoJson('./data/geneva/routes_topo.json', genevaMap.drawRoutes);

// After the routes have been drawn, draw the stops.
genevaMap.on('routesEnd', function () {
    // Load the stops data and pass our drawStops method as the callback to be executed once the data loads.
    genevaDataManager.loadGeoJson('./data/geneva/stops_geo.json', genevaMap.drawStops);
});

// Load the routes data and pass our drawRoutes method as the callback to be executed once the data loads.
sanFranciscoDataManager.loadGeoJson('./data/san_francisco/routes_topo.json', sanFranciscoMap.drawRoutes);
// After the routes have been drawn, draw the stops.
sanFranciscoMap.on('routesEnd', function () {
    // Load the stops data and pass our drawStops method as the callback to be executed once the data loads.
    sanFranciscoDataManager.loadGeoJson('./data/san_francisco/stops_geo.json', sanFranciscoMap.drawStops);
});
```

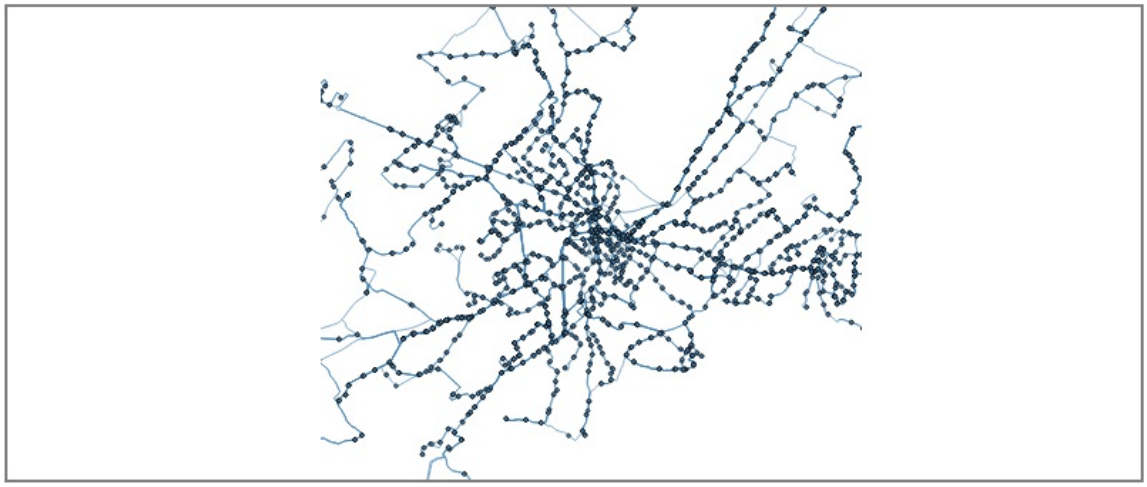
As you can see above, we have instantiated a data manager module, and a mapping module for each one of our cities. We then add an <svg> element for each city, and call the mapping module. As we know, this will grant us access to the <svg> element via our exports function, and allows us to render the routes and stops, once the data loads. Using our data manager module, once we see that we have loaded the data, we call our drawRoutes and drawStops methods, to render the charts.

This code should produce three nicely formatted maps as shown here:

Zurich



Geneva

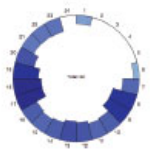


San Francisco

Summary

Now that we have our basic data manager setup, and have produced a map for every city, we need to start making sense of the stop metrics that we have in our data sets. To do so, we are going to want to produce some aggregate statistics that are common among all cities so that we may compare them. With data sets as massive as ours, filtering, grouping, summing, and averaging the data can be a monumental tasks. Fortunately, there are some great libraries out there that make this task much easier for us. We can integrate these libraries into our own custom API to abstract the heavy lifting that these libraries do and make the interface with our module consistent with the rest of the application, as shown in the next chapter.

9. Introducing Crossfilter



IN THIS CHAPTER

- Set up Crossfilter.js for our transit stop data sets
- Define a dimension with Crossfilter.js
- Create a filter method in Crossfilter.js

For our example, we are going to employ Crossfilter.js to make sense of our massive transit stop data sets. We have chosen Crossfilter.js for several reason. First off, it was developed in a large part by Mike Bostock, the developer of D3.js. This means that the patterns used in Crossfilter are very similar to those in D3.js. This makes the code more cohesive and lessens the learning curve. In addition, Crossfilter will allow us to link our visualizations later on as it provides some great filtering methods that we can use to allow for some exploratory analytics. Crossfilter also has some great performance features when working with massive data sets, like ours. This is very important in the browser environment to ensure a pleasant user experience.

Finally, Crossfilter allows us to define our own aggregate functions so that we may investigate metrics that are of interest to us. With minor modifications, we can simply re-filter, or re-aggregate the data with Crossfilter, feed it into our reusable graphing modules, and we will have a completely new dimension that we can evaluate. This is a great feature to have when doing exploratory analytics.

Setting Up Crossfilter

NOTE Source code and data files are available in the code/Chapter09/SettingUpCrossfilter/ directory tree.

There is plenty of documentation to help get you started with using Crossfilter, but we will discuss it as we work through the example. In our application, we want to use Crossfilter to make sense of the transit stop metrics that have been provided to us.

The first step to setup Crossfilter is simply to create a new Crossfilter like so:

```
001: // Define our data manager module.
002: d3Edge.dataManager = function module() {
003:     var exports = {},
004:         dispatch = d3.dispatch('geoReady', 'dataReady', 'dataLoading'),
005:         data,
006:         // Create a new Crossfilter.
007:         transitCrossfilter = crossfilter();
008:     //...
009: }
```

Here, we create a new empty Crossfilter via `transitCrossfilter = crossfilter();`. We assigned this to the local `transitCrossfilter` within our module, which makes it available to us throughout the module. We now have all the Crossfilter methods available on our `transitCrossfilter` variable, but without any data it isn't much use. Adding data to our Crossfilter is quite simple, we simply call:

```
001: // Add data to our Crossfilter.
002: transitCrossfilter.add(data);
```

For our example, we want to add the transit stop metrics after the data has been loaded and cleaned in the browser. If you remember back to our data manager module, we had a method called `loadCsvData` that took care of this for us. So to populate our Crossfilter, we just need to put the code snippet above after our data cleaning in this method and call our `loadCsvData` method.

```
001: // Create a method to load the csv file, and apply cleaning function asynchronously.
002: exports.loadCsvData = function(_file, _cleaningFunc) {
003:     // Create the csv request using d3.csv.
004:     var loadCsv = d3.csv(_file);
005:     // On the progress event, dispatch the custom dataLoading event.
006:     loadCsv.on('progress', function() {
007:         dispatch.dataLoading(d3.event.loaded);
008:     });
009:     loadCsv.get(function (_err, _response) {
010:         // Apply the cleaning function supplied in the _cleaningFunc parameter.
011:         _response.forEach(function (d) {
012:             _cleaningFunc(d);
013:         });
014:         // Assign the cleaned response to our data variable.
015:         data = _response;
016:         // Add data to our Crossfilter.
017:         transitCrossfilter.add(_response);
```

```
018:         // Dispatch our custom dataReady event passing in the cleaned data.
019:         dispatch.dataReady(_response);
020:     });
021: };
022: zurichDataManager.loadCsvData('./data/zurich/zurich_delay.csv', function(d) {
023:     var timeFormat = d3.time.format('%Y-%m-%d %H:%M:%S %p');
024:     d.DELAY = +d.DELAY_MIN;
025:     delete d.DELAY_MIN;
026:     d.SCHEDULED = timeFormat.parse(d.SCHEDULED);
027:     d.LATITUDE = +d.LATITUDE;
028:     d.LONGITUDE = +d.LONGITUDE;
029:     d.LOCATION = [d.LONGITUDE, d.LATITUDE];
030: });
```

Now that we have populated our Crossfilter, let's add a convenience method to our data manager module that will allow us to inspect the size of our Crossfilter. The Crossfilter API provides us with a .size() method that we can invoke to get the number of records in our Crossfilter.

```
001: // Create a convenience method to get the size of our Crossfilter
002: exports.getCrossfilterSize = function () {
003:     return transitCrossfilter.size();
004: };
```

If we invoke this method on the Zurich data manager we should see:

```
001: zurichDataManager.getCrossfilterSize();
002: RETURNS 219371
```

Location Dimension

NOTE The source code and data files are available in the code/Chapter09/LocationDimension/ directory tree.

Now that our Crossfilter has data in it, let's define a dimension. In Crossfilter, a dimension is exactly what it sounds like: a dimension of the data that is of interest to us. We use the dimension to filter on, group on, and compute aggregate statistics on. Each of our data sets has a latitude and longitude field. In our data cleaning function we combined these into a location field and we will create a Crossfilter dimension on this field. To create a dimension, we call the dimension method on our Crossfilter and define an accessor function much like D3.js. First, let us create a local variable, location, in our module that we can assign our dimension to:

```
001: // Define our data manager module.
002: d3Edge.dataManager = function module() {
003:     var exports = {},
004:         dispatch = d3.dispatch('geoReady', 'dataReady', 'dataLoading'),
005:         data,
006:         // Instantiate a new Crossfilter.
007:         transitCrossfilter = crossfilter(),
008:         // Define a location variable for our location dimension.
009:         location;
010:     //..........
011: };
```

Now, we can create our dimension after our data loads:

```
001: // Create a method to load the csv file, and apply cleaning function asynchronously.
002: exports.loadCsvData = function(_file, _cleaningFunc) {
003:     // Create the csv request using d3.csv.
004:     var loadCsv = d3.csv(_file);
005:     // On the progress event, dispatch the custom dataLoading event.
006:     loadCsv.on('progress', function() {
007:         dispatch.dataLoading(d3.event.loaded);
008:     });
009:     loadCsv.get(function (_err, _response) {
010:         // Apply the cleaning function supplied in the _cleaningFunc parameter.
011:         _response.forEach(function (d) {
012:             _cleaningFunc(d);
013:         });
014:         // Assign the cleaned response to our data variable.
015:         data = _response;
016:         // Add data to our Crossfilter.
017:         transitCrossfilter.add(_response);
018:         // Setup the location dimension.
019:         location = transitCrossfilter.dimension(function (d) {
020:             return d.LOCATION;
021:         });
022:        // Dispatch our custom dataReady event passing in the cleaned data.
023:        dispatch.dataReady(_response);
024:     });
025: };
```

In the code above, once our data has loaded, we define our accessor function for our location dimension. This will create a crossfilter dimension using the LOCATION key our data set.

Now that we have defined our dimension, we can filter on it. This is how we can link our two graphics together. Since the map data has a location dimension and our stop data has a location dimension, we can easily filter our stop data based on selected stops on the map. We can create a filter method on our data manager module to accomplish this task. This method will accept an area as its argument. This area will be a geographic box that we will construct using a brush later on in the application. For now we can create the method and pass in some test locations to prove functionality.

Location Filter

NOTE The source code and data files are available in the code/Chapter09/LocationFilter/ directory tree.

For our location filter we know that we want to pass in a geographic bounding box that can be used to filter the stop data. This box will be defined by a longitude and latitude for the top left corner plus a longitude and latitude for the bottom right corner. Any stops that have coordinates within this box will be returned by our filter function. We will use an array of arrays in JavaScript to represent this box. The first element of the array will be an array with coordinates for the top left corner and the second element of the array will be an array with the coordinates for the bottom right corner. We will call the filterFunction on our location dimension and return all records whose coordinates are within our bounding box. For our filter function, the accessor will receive the location array of our dimension. It will look like this:

```
001: // Create a filterLocation method to filter stop data by location area.
002: exports.filterLocation = function (_locationArea) {
003:     // Get the longitudes of our bounding box, and construct an array from them.
004:     var longitudes = [_locationArea[0][0], _locationArea[1][0]],
005:         // Get the latitudes of our bounding box, and construct an array from them.
006:         latitudes = [_locationArea[0][1], _locationArea[1][1]];
007:     location.filterFunction(function (d) {
008:         return d[0] >= longitudes[0]
009:             && d[0] <= longitudes[1]
010:             && d[1] >= latitudes[0]
011:             && d[1] <= latitudes[1];
012:     });
013:     // Return all records within our bounding box.
014:     return location.top(Infinity);
015: };
```

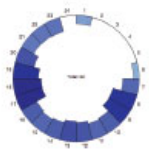
This method will first filter our location dimension by returning the stops within our bounding box, then it returns all of the records by calling top(Infinity). To test this method, let's invoke it passing in a bounding box of the entire world. This should return an array with a length equal to the result of our getCrossfilterSize method.

```
001: zurichDataManager.filterLocation([[-180, -90], [180, 90]]);
002: // Returns Array[219371]
```

Summary

Now that we can filter our stop data by location, we finally have a mechanism to link our two visualizations together. We need to be able to select a geographic bounding area on the map that can be passed into our filter function to return all of the stops within that area. For this we are going to use D3.js's *brush* module. This will allows us to easily select an area on the screen and translate that area into geographic coordinates, as covered in the next chapter.

10. Adding Brushing



IN THIS CHAPTER

- Examine the concept of brushing
- Incorporate brushing into our application

NOTE The source code and data files are available in the code/Chapter10/ directory tree.

Incorporating Brushing into our App

Brushing allows us to interact with our maps by selecting an area on the map, and translating that area to geographic coordinates. This interaction is a key to exploratory data analytics, and will really add a nice touch to each one of our maps by providing a dynamic experience to the user. We will use brushing to select an area on our map, and use our filterLocation method to filter our stop data based on this area. Later on, we will use these filtered locations to inspect the data associated with stop we have selected. the To accomplish this, we will need to modify our map module.

The brush generator in d3.js requires an x and y scale. For our application, these are longitude and latitude respectively. Let's take a look at getting this setup. Our projection generator that we defined earlier scales, centers, and translates our projection within our specified dimensions. For our brush we need to reverse this and get the longitude and latitude coordinate of our specified dimensions so that we can create a scale for both the x and y of the brush generator. Fortunately, d3.js makes this very easy by providing us with an invert method for our projection generator. This method takes an array of pixel coordinates as its argument and returns the longitude and latitude of that point. We can use this method to obtain the minimum and maximum longitude of the x dimension and the minimum and maximum of the y dimension. We will define this in the exports function of our map module. Let us see how this will work:

```
001: function exports(_selection) {
002:     // Set svg equal to the selection that invokes this module.
003:     svg = svg || _selection;
004:     // Bind an empty datum to the selection. Useful later for zooming.
005:     svg.datum([]);
006:     // Set the projection up using our scale, center, and size parameters.
007:     projection = projection || d3.geo.mercator()
008:     .scale(scale)
009:     .center(center)
010:     .translate([size[0]/2, size[1]/2]);
011:     // Set the path up using our projection definied above.
012:     path = path || d3.geo.path()
013:     .projection(projection);
014:     // Get the longitude of the top left corner of our map area.
015:     long1 = projection.invert([0, 0])[0];
016:     // Get the longitude of the top right corner of our map area.
017:     long2 = projection.invert([width, 0])[0];
018:     // Get the latitude of the top left corner of our map area.
019:     lat1 = projection.invert([0, 0])[1];
020:     // Get the latitude of the bottom left corner of our map area.
021:     lat2 = projection.invert([0, height])[1];
022: }
```

Now, we have the longitude and latitude of the extents of our map. We can use these to create linear scales that can be used for the x, and y scales of our brush. To do this, we will use the d3.js linear scale generator, using the size of our map as the range, and our computed longitudes and latitudes as the domain.

```
001: //Create a linear scale generator for the x of our brush.
002: brushX = d3.scale.linear()
003:   .range([0, size[0]])
004:   .domain([long1,long2]);
005:
006: //Create a linear scale generator for the y of our brush.
007: brushY = d3.scale.linear()
008:   .range([0, size[1]])
009:   .domain([lat1, lat2]);
```

We now have an x and a y scale that we can use to create our brush. Creating the brush is rather simple:

```
001: //Create our brush using our brushX and brushY scales.
002: brush = d3.svg.brush()
003:   .x(brushX)
004:   .y(brushY)
005:   .on('brush', function () {dispatch.brushing(brush);});
```

The Created Brush

Putting it all together, we end up with the following:

```
//Create and exports function that can be invoked on a selection.
function exports(_selection) {

    //Set svg equal to the selection that invokes this module.
    svg = svg || _selection;

    //Bind an empty datum to the selection. Useful later for zooming.
    svg.datum([]);

    //Set the projection up using our scale, center, and size parameters.
    projection = projection || d3.geo.mercator()
      .scale(scale)
      .center(center)
      .translate([size[0]/2, size[1]/2]);

    //Set the path up using our projection defined above.
    path = path || d3.geo.path()
      .projection(projection);

    //Get the longitude of the top left corner of our map area.
    long1 = projection.invert([0,0])[0];
    //Get the longitude of the top right corner of our map area.
    long2 = projection.invert([width, 0])[0];

    //Get the latitude of the top left corner of our map area.
    lat1 = projection.invert([0,0])[1];
    //Get the latitude of the bottom left corner of our map area.
    lat2 = projection.invert([width, height])[1];

    //Create a linear scale generator for the x of our brush.
    brushX = d3.scale.linear()
      .range([0, size[0]])
      .domain([long1,long2]);

    //Create a linear scale generator for the y of our brush.
    brushY = d3.scale.linear()
      .range([0, size[1]])
      .domain([lat1, lat2]);

    //Create our brush using our brushX and brushY scales.
    brush = d3.svg.brush()
      .x(brushX)
      .y(brushY)
      .on('brush', function () {dispatch.brushing(brush);});

}
```

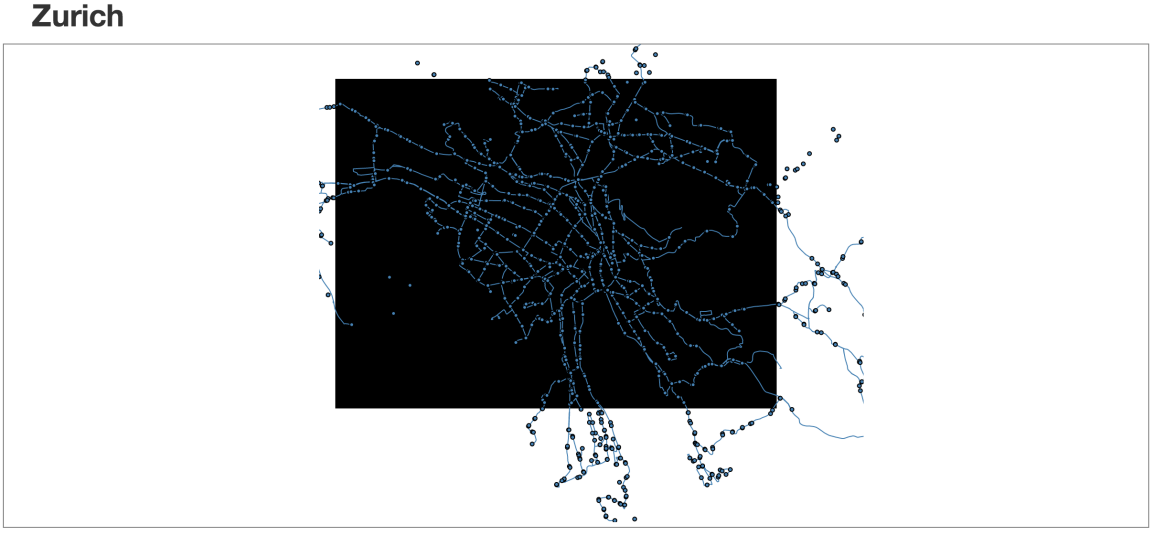
In this method we have created the brush using our brushX and brushY as the x and y scales. In addition we have dispatched a custom event called brushing and passed in the brush so that we may access it on our map module. We will use this to get the extents of our brush and pass them into the filter function we defined earlier. All that remains to create the brush is to create a method that will append to our svg element:

```
//Create our addBrush method.
exports.addBrush = function () {
    svg.append('g')
    .attr('class', 'brush')
    .call(brush)
    .selectAll('rect')
    .attr('width', width);
    return this;
};
```

Now we can invoke this method on our map module for Zurich and inspect the extent of our brush by listening to the custom brushing event we defined earlier:

```
zurichMap.addBrush()
.on('brushing', function (brush) {
    console.log(brush.extent());
});
```

If we navigate to our application in the browser, when we hover over the map of Zurich, we will see crosshairs indicating that we can brush over the area. Brushing over the map will create a black area over the map like this:

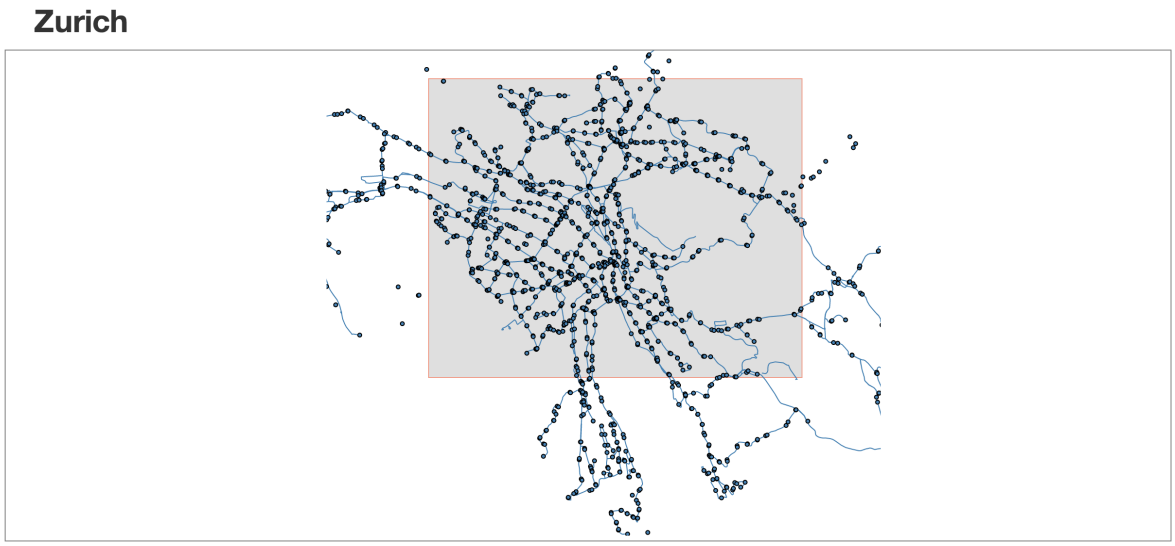


The Created Brush

Let's add some styling to make this look a little better:

```
.brush .extent {
    stroke: #f09f8c;
    fill-opacity: .125;
    shape-rendering: crispEdges;
}
```

Now when we brush over the map of Zurich, we will get a nicely formatted box like this:



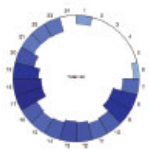
If we inspect the console of our browser, we will see an array every time the brush changes. This array contains two more arrays, one for the top left corner of our brushed area and one for the bottom right of our brushed area. Each one of these arrays contains the longitude and latitude of these respective points:

```
[ [ 8.468999999999998, 47.40062403830741 ], [ 8.468999999999998, 47.40062403830741 ] ]
```

Summary

This is the same structure that we passed into our filterLocation method on our data manager to return all of the stops within the area. This is what we will use to link our two visualizations together as it will allow us to filter the stop data that we will use for our radial histogram simply by brushing over an area of the map. We will develop our radial histogram in Chapter 12 but first, in the next chapter, we need to attend to integrating Crossfilter properly into the application.

11. Integrating Crossfilter



IN THIS CHAPTER

- Use Crossfilter to filter our stop metric data
- Create aggregate statistics on these filtered locations

NOTE The source code and data files are available in the code/Chapter11/ directory tree.

Now that we have our brushing setup for our maps and it returns an array that can be accepted by the filterLocation filter we defined earlier, we can easily integrate the two so that brushing over the map will filter our stop metric data. We can do this simply by passing in the extents of our brush, obtained by brush.extent(), into our filter function.

```
001: zurichMap.on('brushing', function (brush) {
002:     var filteredLocations = zurichDataManager.filterLocation(brush.extent());
003: });
```

Every time we brush over the map, the extents of our brush will be passed into our filter function and assigned to the local variable filteredLocations. We can now perform some aggregate statistics on these filtered locations and pass them into the radial histogram we will develop in the next chapter. Let's take a look how we will generate these statistics.

Just the same way we created a method in our data manager module to filter by locations, we will create another method that we can use to filter and aggregate the data by stop delays. We will setup our method like so:

```
001: // Create a getDelays method to filter and aggregate the delay data by stop.
002: exports.getDelay = function (_locations) {
003:     //...
004: };
```

Filtering with Crossfilter

For this method we are going to pass in our filtered locations as the only argument. Since this is just an array of locations, we can create another Crossfilter within our new method to filter, group, and reduce on. We do this since we want Crossfilter to remain local to this method. Otherwise, if we used a Crossfilter defined at the top of our module definition, every time we brushed over the map we would be adding data to the Crossfilter. By defining it locally, we ensure that the Crossfilter only contains data for the area that we have selected. The code looks like this:

```
001: // Create a getDelays method to filter and aggregate the delay data by stop.
002: exports.getDelay = function (_locations) {
003:     var delayCrossfilter = crossfilter(),
004:     // Create a dimension by hour of the scheduled stop time.
005:     stopTime = delayCrossfilter.dimension(function (d) {
006:         return d.SCHEDULED.getHours();
007:     });
008:     // Group the dimension by hour, and reduce it by increasing the count for all delays greater than 1.
009:     lateArrivalsByStopTime = stopTime.group().reduce(
010:         function reduceAdd(p, v) {
011:             return v.DELAY > 0 ? p + 1 : p + 0;
012:         },
013:         function reduceRemove(p,v) { return 0; },
014:         function reduceInitial(p,v) { return 0; }
015:     );
016:     // Add our filtered locations to the crossfilter.
017:     delayCrossfilter.add(_locations);
018:     // Return an array contained the aggregated data, and the stopTime dimension.
019:     return [stopTime, lateArrivalsByStopTime];
020: };
```

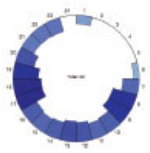
What this method is doing is creating a dimension based upon the hour of the scheduled stop time. It is then grouping all of the selected stops by this dimension and then reducing all of these records to a single value for each hour. This value is computed simply by incrementing the count if the delay is greater than one. What we will end up with is a count for every hour containing all of the transit stops events that were delayed for a given set of locations. The reason we return an array containing both the group and the dimension is we will want access to both of these later when we construct our radial histogram. Now we can inspect the data like this:

```
001: zurichMap.on('brushing', function (brush) {
002:     var filteredLocations = zurichDataManager.filterLocation(brush.extent());
003:     delaysByHourAndLocation = zurichDataManager.getDelays(filteredLocations);
004:     // Inspect the total number of events by hour;
005:     console.log(delaysByHourAndLocation[0].group().all());
006:     // Inspect the total number of delays by hour.
007:     console.log(delaysByHourAndLocation[1].all());
008: });
```

Summary

Now, when you brush over the map you should see the data in the console for the area you have selected. Our next and final step is to develop the radial histogram to visualize this data and update it every time we brush over the map, shown in the next chapter.

12. Radial Histogram API

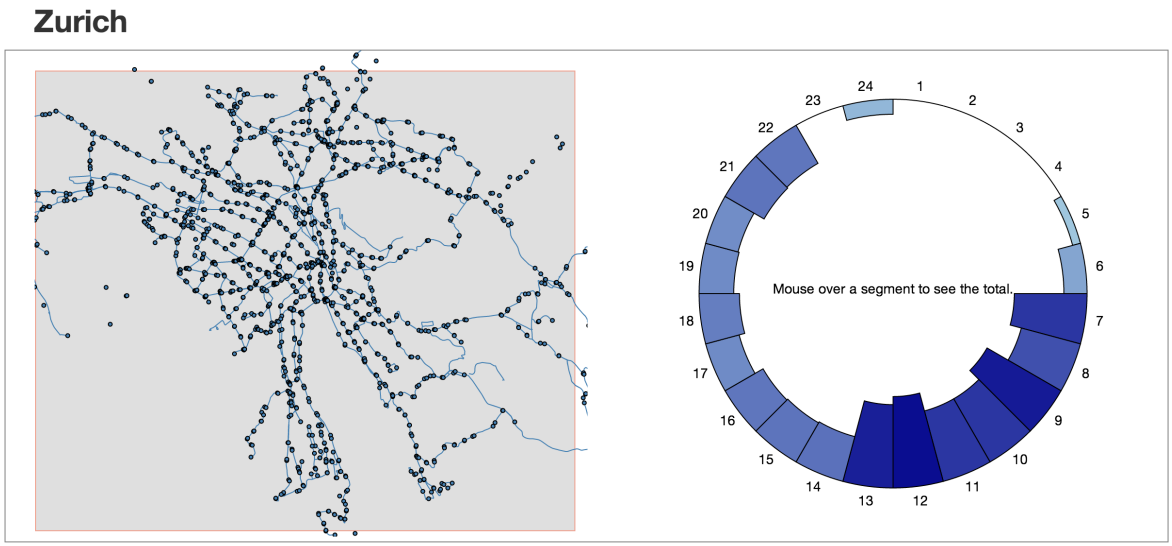


IN THIS CHAPTER

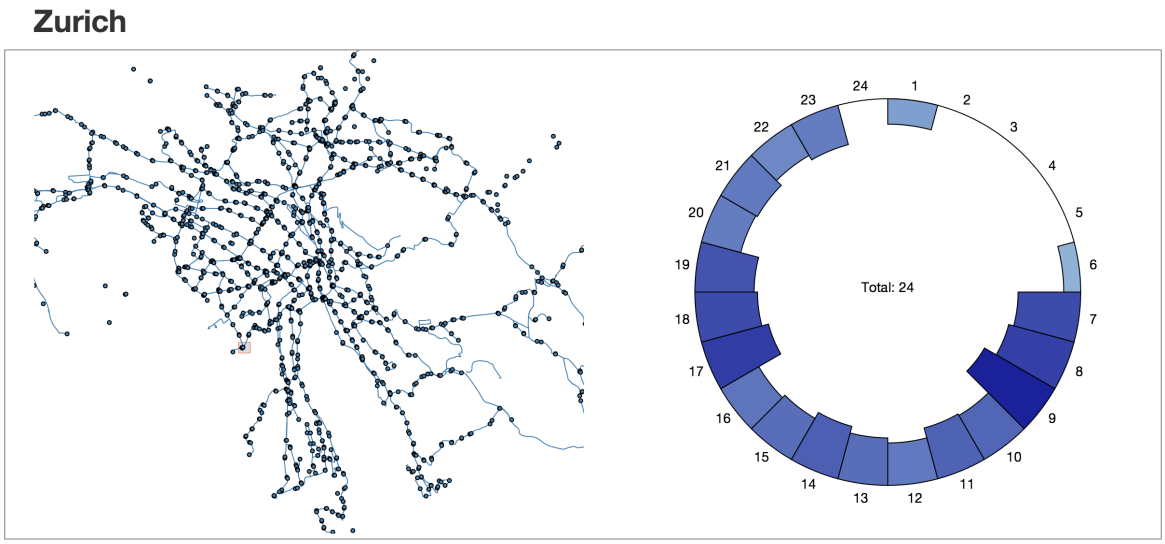
- Understand the Radial Histogram API
- Implement the Radial Histogram into our application

NOTE The source code and data files are available in the code/Chapter12/ directory tree.

The idea for a radial histogram came about after grappling with the best way to simultaneously visualize a time dimension and an aggregation of the stop metrics we were provided in our example. Because time is cyclical, a radial layout allows us to "bin" the data easily into 24 segments, one for each hour of the day. Each segment's inner radius is proportional to the value of the metric we pass into our module. The metric of most interest for us was the number of times the bus, or buses, were delayed in each individual hour. This allows us to get a picture of peak hours very quickly. The end result looks something like this:



With this visualization we are able to quickly tell when and where most of the delays occur. By brushing over the majority of a map we can get an overall snapshot of when the delays occur for all stops. In the image above it is very apparent that the majority of the delays for all of Zurich happen between the hours of 7 AM and 1 PM. This visualization allows us to pinpoint specific areas of the city with the brush. Take the following example:



Making the Visualization Reusable

In this example we have brushed over a very small region of Zurich, encompassing only a few stops. The radial histogram here shows that the majority of the delays happen during rush hour, suggesting that perhaps these few stops are mostly used by work commuters. Let's see how we made this visualization reusable.

Just like with our map module we setup the defaults, the selection function, and the *getters / setters*:

```
001: d3Edge.radialHistogram = function module () {
002:     var slices = 24, // 24 hours in a day.
003:         innerRadius = 100, // Default inner radius
004:         outerRadius = 300, // Default outer radius
005:         innerScale = d3.scale.linear(), // Define a scale for sizes segments based on value.
006:         group, // Our empty group variable
007:         dimension, // Our empty dimension variable.
008:         offset = 50, // Label offset value.
009:         lowerColor, // The color used for the minimum of our range
010:         upperColor, // The color used for the maximum of our range
011:         innerRange, // The lower bound for radius value
012:         outerRange, // The upper bound for radius value
013:         color = d3.scale.linear(); // Linear color scale used for the segments.
014:     // The chart function our module will return with the selection that called it,
015:     // as the only argument.
016:     function chart (_selection) {
017:     }
018:     // Method to get/set the inner radius.
019:     chart.innerRadius = function (_innerRadius) {
020:         if (!arguments.length) return innerRadius;
021:         innerRadius = _innerRadius;
022:         return chart;
023:     };
024:     // Method to get/set the outer radius.
025:     chart.outerRadius = function (_outerRadius) {
026:         if (!arguments.length) return outerRadius;
027:         outerRadius = _outerRadius;
028:         return chart;
```

```
    };
    // Method to get/set the crossfilter group.
    chart.group = function (_group) {
        if (!arguments.length) return group;
        group = _group;
        return chart;
    };
    // Method to get/set the label offset.
    chart.offset = function (_offset) {
        if (!arguments.length) return offset;
        offset = _offset;
        return chart;
    };
    // Method to get/set the crossfilter dimension.
    chart.dimension = function (_dimension) {
        if (!arguments.length) return dimension;
        dimension = _dimension;
        return chart;
    };
    // Method to get/set the color range.
    chart.colorRange = function (_array) {
        if (!arguments.length) return [lowerColor, upperColor];
        lowerColor = _array[0];
        upperColor = _array[1];
        return chart;
    };
    // Method to get/set the radial range.
    chart.radialRange = function (_array) {
        if (!arguments.length) return [innerRange, outerRange];
        innerRange = _array[0];
        outerRange = _array[1];
        return chart;
    };
    // Finally, return the chart.
    return chart;
};
```

Creating the Visualization

This takes care of the basics for our module. Now we just need to work on the code that will execute on the d3.js selection and actually create the visualization. This is the code that will be executed every time we call our module on the selection, and thus is also the code responsible for updating the visualization when new data is passed in.

The first part of our function is pretty straightforward. We first look to see if the inner and outer ranges have been set by our setter functions, and if not we set them equal to the default inner and outer radius, respectively. Then we use the d3.js arc generator to construct a generator for both segments of the radial chart and the labels.

```
function chart (_selection) {
    // If the innerRange is not defined, it equals the innerRadius.
    innerRange = innerRange ? innerRange : innerRadius;
    // If the outerRange is not defined, it equals the outerRadius.
    outerRange = outerRange ? outerRange : outerRadius;
    // Our d3 arc generator for the segments.
    var arc = d3.svg.arc()
    .innerRadius(function (d, i) { return innerScale(d); })
    .outerRadius(function (d, i) { return outerRadius; })
    .startAngle(function (d, i) {
        return 2 * Math.PI * (i / slices);
    })
    .endAngle(function (d, i) {
        return 2 * Math.PI * ((i + 1) / slices);
    });
    // Our d3 arc generator for the labels.
    var label = d3.svg.arc()
    .innerRadius(outerRadius + offset)
    .outerRadius(outerRadius + offset)
    .startAngle(function (d, i) {
        return 2 * Math.PI * (i / slices);
    })
    .endAngle(function (d, i) {
        return 2 * Math.PI * ((i + 1) / slices);
    });
}
```

The code above setups our arc generator by setting the inner and out range of the radial histogram to the value specified in the setter functions, and also sets the angle of the arcs based on how many slices are defined in the setter function. In addition, it sets up an arc function for the labels. The label function is essentially the same, but we add an offset value to offset the label text from the actual graphic. This offset is also defined via a setter function.

The next step is to compute the length and the minima and the maxima of the data we will pass into our module via the dimension and group methods. We need the length of both arrays to see if we need to add hours to the beginning or the end of the array. If there are no records for an hour in Crossfilter, these hours will not be included in the resulting array. This is undesirable, since it will shift all of the hours around when they are entered into the DOM via the enter() command. The dimension method gives us access to all of the records in our Crossfilter dimensions. In our case it will return all of the delays, both positive and negative. A negative delay is simply an early arrival. We need this information so that our data will be scaled to the total number of records. The group method gives us access to all of the records that are considered to be delayed. This is the data that will comprise our visualization. Let's add this code:

```
function chart (_selection) {
    // If the innerRange is not defined, it equals the innerRadius.
    innerRange = innerRange ? innerRange : innerRadius;
    // If the outerRange is not defined, it equals the outerRadius.
    outerRange = outerRange ? outerRange : outerRadius;
    // Our d3 arc generator for the segments.
    var arc = d3.svg.arc()
    .innerRadius(function (d, i) { return innerScale(d); })
    .outerRadius(function (d, i) { return outerRadius; })
    .startAngle(function (d, i) {
        return 2 * Math.PI * (i / slices);
```

```
    })
    .endAngle(function (d, i) {
        return 2 * Math.PI * ((i + 1) / slices);
    });
    //Our d3 arc generator for the labels.
    var label = d3.svg.arc()
    .innerRadius(outerRadius + offset)
    .outerRadius(outerRadius + offset)
    .startAngle(function (d, i) {
        return 2 * Math.PI * (i / slices);
    })
    .endAngle(function (d, i) {
        return 2 * Math.PI * ((i + 1) / slices);
    });
    // The total number of records for the city
    var totalRecords = dimension.group().all(),
        // The total number of delays for they city.
        totalDelays = group.all();
    // Obtain the min and max for both totalRecords and totalDelays.
    // if there are no records, set to zero.
    var mintotalRecords = totalRecords.length
        ? +totalRecords[0].key : 0,
        maxtotalRecords = totalRecords.length
        ? +totalRecords[totalRecords.length - 1].key : 0,
        mintotalDelays = totalDelays.length
        ? +totalDelays[0].key : 0,
        maxtotalDelays = totalDelays.length
        ? +totalDelays[totalDelays.length - 1].key : 0;
    // We must always have an array of length 24. Inspect the totalRecords array,
    // and totalDelays array and splice to the beginning and end as required.
    for (i = 0; i < mintotalRecords; i++) {
        totalRecords.splice(i, 0, {key: i, value: 0});
    }
    for (i = maxtotalRecords; i < 24; i++) {
        totalRecords.splice(i, 0, {key: i, value: 0});
    }
    for (i = 0; i < mintotalDelays; i++) {
        totalDelays.splice(i, 0, {key: i, value: 0});
    }
    for (i = maxtotalDelays; i < 24; i++) {
        totalDelays.splice(i, 0, {key: i, value: 0});
    }
    // Get the min and max values for both totalRecords, and totalDelays. We
    // will use this for our scales.
    var totalRecordsMax = d3.max(totalRecords, function (d) {
        return d.value;
    }),
    totalRecordsMin = d3.min(totalRecords, function (d) {
        return d.value;
    });
}
```

While the code above looks intimidating, we are simply inspecting the data we want to visualize, and checking for missing hours in the data set. If an hour is missing, we simply splice it into the data set with a value of 0, to ensure that our radial histogram accurately displays the information.

DOM Manipulation

Now all that is left to do is handle the DOM manipulation. This is just standard d3.js that we have become accustomed to. First, we set the range and domain of both our innerScale and colorScale using the current data. Then, we select any existing arcs and update them with the new data. If any new arcs are required, we enter them into the DOM. If any arcs are no longer required, we remove them from the DOM. Then we add our mouseover listener to all of the arcs and add our labels. The final code looks like this:

```
function chart (_selection) {
    // If the innerRange is not defined, it equals the innerRadius.
    innerRange = innerRange ? innerRange : innerRadius;
    // If the outerRange is not defined, it equals the outerRadius.
    outerRange = outerRange ? outerRange : outerRadius;
    // Our d3 arc generator for the segments.
    var arc = d3.svg.arc()
    .innerRadius(function (d, i) { return innerScale(d); })
    .outerRadius(function (d, i) { return outerRadius; })
    .startAngle(function (d, i) {
        return 2 * Math.PI * (i / slices);
    })
    .endAngle(function (d, i) {
        return 2 * Math.PI * ((i + 1) / slices);
    });
    // Our d3 arc generator for the labels.
    var label = d3.svg.arc()
    .innerRadius(outerRadius + offset)
    .outerRadius(outerRadius + offset)
    .startAngle(function (d, i) {
        return 2 * Math.PI * (i / slices);
    })
    .endAngle(function (d, i) {
        return 2 * Math.PI * ((i + 1) / slices);
    });
    // The total number of records for the city
    var totalRecords = dimension.group().all(),
        // The total number of delays for they city.
        totalDelays = group.all();
    // Obtain the min and max for both totalRecords and totalDelays.
    // if there are no records, set to zero.
    var mintotalRecords = totalRecords.length
        ? +totalRecords[0].key : 0,
        maxtotalRecords = totalRecords.length
        ? +totalRecords[totalRecords.length - 1].key : 0,
        mintotalDelays = totalDelays.length
        ? +totalDelays[0].key : 0,
        maxtotalDelays = totalDelays.length
        ? +totalDelays[totalDelays.length - 1].key : 0;
    // We must always have an array of length 24. Inspect the totalRecords array,
    // and totalDelays array and splice to the beginning and end as required.
    for (i = 0; i < mintotalRecords; i++) {
        totalRecords.splice(i, 0, {key: i, value: 0});
    }
    for (i = maxtotalRecords; i < 24; i++) {
```

```
        totalRecords.splice(i, 0, {key: i, value: 0});
    }
    for (i = 0; i < mintotalDelays; i++) {
        totalDelays.splice(i, 0, {key: i, value: 0});
    }
    for (i = maxtotalDelays; i < 24; i++) {
        totalDelays.splice(i, 0, {key: i, value: 0});
    }
    // Get the min and max values for both totalRecords, and totalDelays. We
    // will use this for our scales.
    var totalRecordsMax = d3.max(totalRecords, function (d) {
        return d.value;
    }),
        totalRecordsMin = d3.min(totalRecords, function (d) {
        return d.value;
    });
    // Set the range and domain for our innerScale using the min and max from the totalRecords.
    innerScale.range([outerRange, innerRange]).domain([totalRecordsMin, totalRecordsMax]);
    // Set the color range similarly
    color.range([lowerColor, upperColor]).domain([totalRecordsMin, totalRecordsMax]);
    // Update our segments using the current data.
    var arcs = _selection.selectAll('path')
    .data(totalDelays)
    .attr('d', function (d,i) { return arc(d.value,i); })
    .attr('fill', function (d) { return color(d.value); })
    .attr('stroke', 'black')
    .attr('class', 'slice');
    // Add any new segments using the current data.
    arcs.enter().append('path')
    .attr('d', function (d,i) {return arc(d.value,i);})
    .attr('fill', function (d) {return color(d.value);})
    .attr('class', 'slice')
    .attr('stroke', 'black');
    // Remove and extra segments.
    arcs.exit().remove();
    // Attach our mouseover event.
    arcs.on('mouseover', mouseover);
    // Add our labels.
    var labels = _selection.selectAll('text')
    .data(totalDelays).enter()
    .append("text")
    .attr("transform", function(d,i) {
        return "translate(" + label.centroid(d, i) + ")";
    })
    .attr("dy", ".35em")
    .attr("text-anchor", "middle")
    .text(function(d,i) { return i + 1; });
    // Remove center text on chart update.
    _selection.selectAll('.centerText').remove();
    // Add the center text for the chart.
    var centerText = _selection.append('text')
    .attr('text-anchor', 'middle')
    .text('Mouse over a segment to see the total.')
    .attr('class', 'centerText');
    // On mouseover function to display segment total.
    function mouseover (d) {
        centerText.text('Total: ' + d.value);
    }
}
```

The code above is where we bind our data to the DOM, and the enter, exit, or update as needed. This should be a very familiar pattern to you if you have been using D3.js.

Our Final App Visualizations

Our reusable module is set up. Now to invoke it, we just need to instantiate it with our desired options and call it on a d3.js selection:

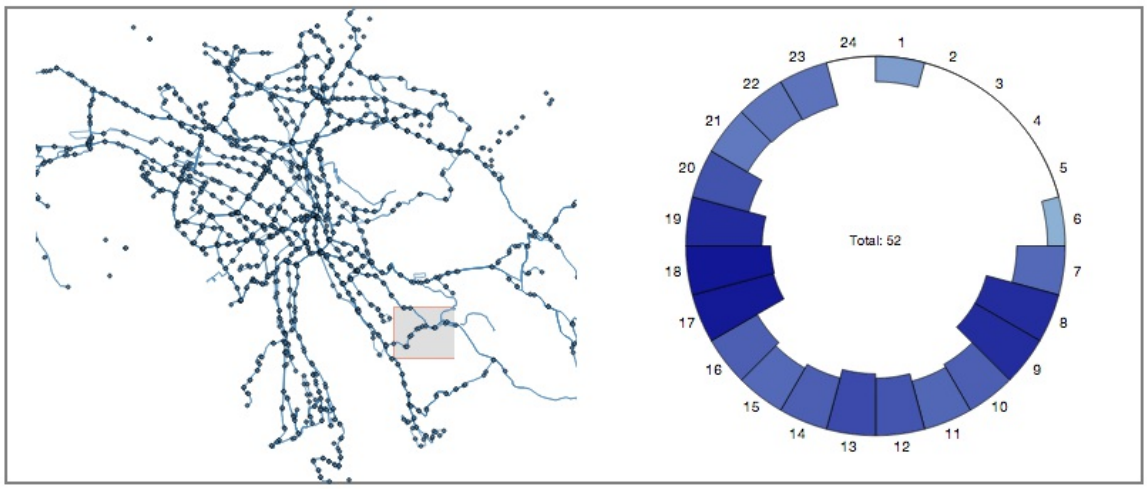
```
// Instantiate our radial module for each city.
var zurichRadial = d3Edge.radialHistogram()
.colorRange(['lightblue', 'darkblue'])
.innerRadius(5)
.outerRadius(200)
.offset(15)
.radialRange([100, 200]);
// Set up the DOM for each city for the radial chart.
var zurichHist = d3.select('#zurich_hist')
.append('svg')
.attr('width', width)
.attr('height', height)
.append('g')
.attr('transform', 'translate(' + width/2 + ',' + height/2 + ')');
// Call our module.
zurichHist.call(zurichRadial);
```

This will initialize our visualization with no data, so we aren't going to see much. We need to pass in the data we filtered by brushing over the map. We will do this by using the dimension and group methods of our module. We will need to do this inside the brushing event handler of our mapping module, just like we did at the end of the last chapter.

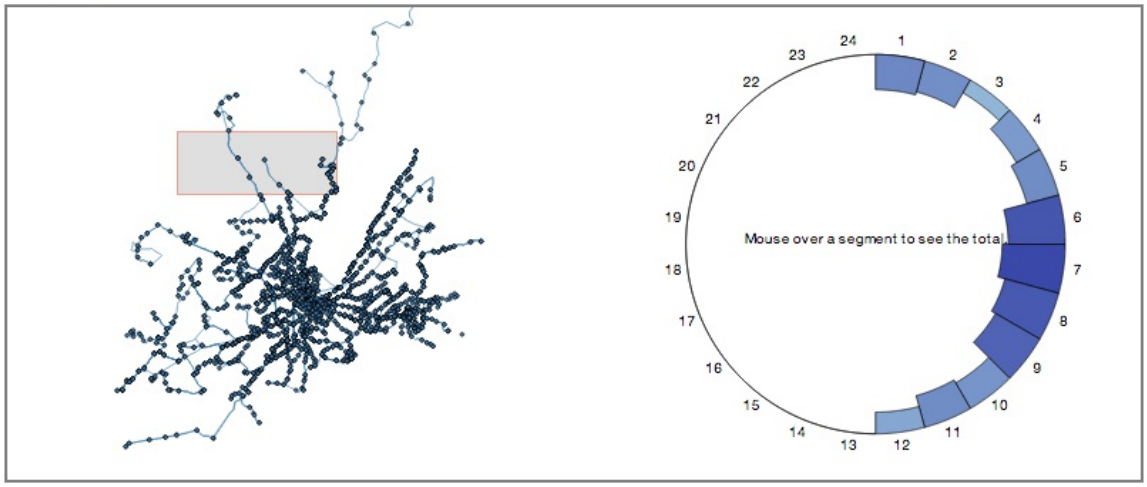
```
// On map brushing, filter the stop metric data, pass filtered data into radial chart.
zurichMap.on('brushing', function (brush) {
    // Get the locations inside the brush.
    var filteredLocations = zurichDataManager.filterLocation(brush.extent()),
    // Get the delays inside the area.
    delaysByHourAndLocation = zurichDataManager.getDelays(filteredLocations);
    // Pass in our filtered delays to the radial histogram.
    zurichRadial.group(delaysByHourAndLocation[1]).dimension(delaysByHourAndLocation[0]);
    // Update radial chart with the new data.
    zurichHist.call(zurichRadial);
});
```

Now every time we brush over the map, we will re-filter our data and re-render our radial histogram! Let's take a final look at the application with all three visualizations:

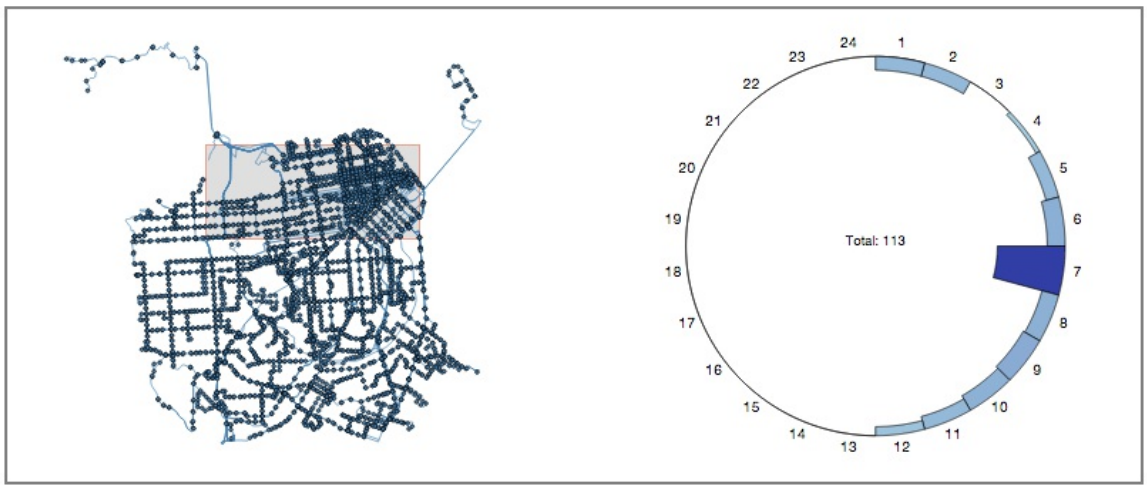
Zurich



Geneva



San Francisco



There you have it! A comprehensive, interactive visualization of three cities using reusable components!

Summary

This application serves to demonstrate the power of the reusable API. We have created a complex visualization consisting of brushing, custom events, crossfiltering, and dynamic updating, but we have done so in a way that is easily adaptable and scalable. This application could be adapted quickly to use the data from your own city. We encourage you to take the source code and try it out!

13. Conclusion

We began this book with a quick look at a standard approach to writing a D3 chart. If you've reached this far in the book, and you've read it all, nicely done! Hopefully we've opened your eyes to a much more sustainable and reusable method to using D3.js, and how powerful D3.js applications can be.

We had a number of intentions in mind when writing this book:

- Give greater understanding to some of the core patterns that lie at the very heart of D3.js.
- Provide documentation and examples of how D3.js code could be written in a more reusable and sharable fashion.
- Encourage you, the reader, to do this yourself, whenever you write D3.js code.
- Help those keen to move on from a basic level of D3.js to a more intermediate-advanced level of understanding.

We very much hope that we've achieved them!

Where to download the code listed in this book

As mentioned in the Introduction, you can get all the source code, data files, etc. on github: https://github.com/backstopmedia/D3Edge. You can either download the latest version of all the files in a ZIP archive at the URL below or use git to fetch the repository: https://github.com/backstopmedia/D3Edge/archive/master.zip.

Useful Sources of Information:

- Mike Bostock: http://bl.ocks.org/mbostock/
- Jason Davies: https://github.com/jasondavies
- D3 Wiki: https://github.com/mbostock/d3/wiki
- D3 on StackOverflow: http://stackoverflow.com/questions/tagged/d3.js
- List of D3 Plugins: https://github.com/d3/d3-plugins
- Meetups:
  - London: http://lanyrd.com/series/london-d3-js/
  - SF: http://www.meetup.com/Bay-Area-d3-User-Group/
  - Texas: http://www.meetup.com/Austin-d3-js-Meetup/

Where You Can Find Us

Christophe Viau: https://twitter.com/d3visualization

Roland Dunn: https://twitter.com/roland_dunn

Andy Thornton: https://twitter.com/graftdata

Ger Hobbelt: https://twitter.com/Ger_Hobbelt
(though I very much prefer email: ger@hobbelt.com)

Troy Mott: http://bleedingedgepress.com - troy@backstopmedia.com

Of course you can also find us at the D3 mailing list. Though if you have a question about the book or issues with the provided sourcecode you may also file an 'issue' at the book's github site here: https://github.com/backstopmedia/D3Edge/issues.

We all very much hope you've found the book useful, and look forward to seeing how you build and use D3 in the future!