7

Depending on the Open Source Sphere

In this chapter we will cover:

- Animating a gauge meter (jqPlot)
- Creating an animated 3D chart (canvas3DGraph)
- Charting over time (flotJS)
- Building a clock with RaphaelJS
- Making a sunburst chart with InfoVis

Introduction

The open source data visualization community is extremely rich and detailed, with many options and some really amazing libraries. Each library has its strong points and its disadvantages. Some are standalone code while others depend on other platforms such as jQuery. Some are really big and some are really small; there isn't any one option that is perfect for all opportunities, but with such a rich amount of options, the most important thing is to figure out what library is the right one for you.

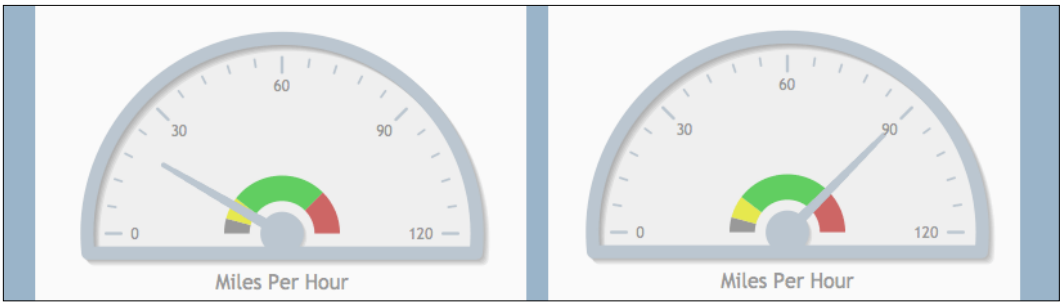
There is always a trade-off when working with open source libraries, mainly when it comes to file sizes and having just too many features that drag down the speed of your application, load time, and so on. But with the richness and creativeness of the community, it's hard to avoid really fantastic charts that can be created in minutes instead of hours.

In this chapter we will explore working with some of these options. Instead of using the libraries according to the documentation of the projects, our goal will be to find ways to override the built-in libraries to provide us with better control over our applications, in case we can't find a suitable solution in the documentation of an application. So the goal in this chapter is now double, namely to find ways to do things that aren't naturally set to work and to find ways to bypass problems.

One more important thing to note is that all of these open source libraries have copyrights. It is advised that you check the legal documentation of the project before you go ahead with it.

Animating a gauge meter (jqPlot)

In this recipe, we will be creating a really fun gauge meter and injecting some random animation into it to make it look like a real source of live data is connected to it, such as the speed of a car:



Getting ready

To get started you will need to use jQuery and jqPlot. This time around we will start from scratch.

To get the latest scripts, visit the creator site at `http://blog.everythingfla.com/?p=339`.

Download both jQuery and jqPlot, or download our source files to start with.

How to do it...

Let's list the steps required to complete the task:

1. Create an HTML page for our project:

```
<!DOCTYPE html>
<html>
  <head>
    <title>JQPlot Meter</title>
```

```
    <meta charset="utf-8" />
    <link rel="stylesheet"
    href="./external/jqplot/jquery.jqplot.min.css">
    <script src="http://ajax.googleapis.com/
    ajax/libs/jquery/1.7.2/jquery.min.js"></script>
    <script src="./external/jqplot/
    jquery.jqplot.js"></script>
    <script src="./external/jqplot/plugins/
    jqplot.meterGaugeRenderer.min.js"></script>

    <script src="./07.01.jqplot-meter.js"></script>
  </head>
  <body style="background:#fafafa">

  <div id="meter" style="height:400px;width:400px;
  "></div>
  </body>
</html>
```

2. Create the `07.01.jqplot-meter.js` file.
3. Let's add a few helper variables. We will use them when rendering our meter:

```
var meter;
var meterValue=0;
var startingSpeed = parseInt(Math.random()*60) + 30;
var isStarting = true;
var renderOptions= {
               label: 'Miles Per Hour',
               labelPosition: 'bottom',
               labelHeightAdjust: -10,
               intervalOuterRadius: 45,
               ticks: [0, 40, 80, 120],
               intervals:[25, 90, 120],
               intervalColors:[ '#E7E658','#66cc66',
               '#cc6666']
            };
```

4. Now it's time to create our meter. We will use jQuery to know when our document is being read and then create our chart.

```
$(document).ready(function(){

  meter = $.jqplot('meter',[[meterValue]],{
    seriesDefaults: {
      renderer: $.jqplot.MeterGaugeRenderer,
      rendererOptions:renderOptions
    }
  });

});
```

5. Now it's time to animate our chart. Let's add in the last line of our `ready` listener interval (it will run from now on until the end of the recipe):

```
$(document).ready(function(){

  meter = $.jqplot('meter',[[meterValue]],{
    seriesDefaults: {
      renderer: $.jqplot.MeterGaugeRenderer,
      rendererOptions:renderOptions
    }
  });

  setInterval(updateMeter,30);

});
```

6. Last but not least, it's time to create the `updateMeter` function:

```
function updateMeter(){
  meter.destroy();


  if(isStarting && meterValue<startingSpeed){
    ++meterValue
  }else{
    meterValue += 1- Math.random()*2;
    meterValue = Math.max(0,Math.min(meterValue,120)); //keep our
value in range no mater what
  }

  meter = $.jqplot('meter',[[meterValue]],{
    seriesDefaults: {
      renderer: $.jqplot.MeterGaugeRenderer,
      rendererOptions:renderOptions
    }
  });

}
```

Well done. Refresh your browser and you will find an animated speedometer that looks like that of a car driving around (if you only imagine it).

How it works...

This task was really easy as we didn't need to start everything from scratch. For the meter to run, we need to import the library `meterGaugeRenderer`. We do that by adding that into our JavaScript files that we are loading. But let's focus on our code. The first step in our JavaScript is to prepare a few global variables; we are using global variables as we want to re-use these variables in two different functions (when we are ready to reset our data).

```
var meter;
var meterValue=0;
var startingSpeed = parseInt(Math.random()*60) + 30;
var isStarting = true;
```

The `meter` variable will hold the meter that we will generate from our open source library. The `meterValue` will be our initial value when the application loads. Our `startingSpeed` variable is going to be a random value between `30` and `90`. The goal is to start from a different place each time to make it more interesting. As soon as our application starts, we will want our meter to quickly animate to its new base speed (the `startingSpeed` variable). Lastly, this connects to the `isStarting` variable as we will want to have one animation that will get us to our base speed. When we get there, we want to switch to a random driving speed that would cause the animation to change. Now that we have all the helper variables set, we are ready to create the `renderOptions` object:

```
var renderOptions= {
                label: 'Miles Per Hour',
                labelPosition: 'bottom',
                labelHeightAdjust: -10,
                intervalOuterRadius: 45,
                ticks: [0, 40, 80, 120],
                intervals:[25, 90, 120],
                intervalColors:[ '#E7E658','#66cc66', '#cc6666']
            };
```

This object is really the heart of the visuals for our application. (There are other options that you are welcome to explore in the jqPlot project home page documentation.) Now let's review a few of the key parameters.

`intervalOuterRadius` has a bit of a tricky name, but it's actually the internal radius. The actual size of our meter is controlled by the size of `div` that we set our application to be in. `intervalOuterRadius` controls the size of our internal shape in the speedometer's core.

```
var renderOptions= {
  label: 'Miles Per Hour',
  labelPosition: 'bottom',
  labelHeightAdjust: -10,
  intervalOuterRadius: 45,
```

```
    //ticks: [0, 40, 80, 120],
    intervals:[10,25, 90, 120],
    intervalColors:['#999999', '#E7E658','#66cc66', '#cc6666']
};
```

The `ticks` function controls where the copy outlines would be. The default would take our top range and divide it by 4 (that is 30, 60, 90, and 120). The `intervals` and `intervalColors` functions let the meter know the ranges and the inner, internal, pie colors (separated from the ticks).

```
$(document).ready(function(){

  meter = $.jqplot('meter',[[meterValue]],{
    seriesDefaults: {
      renderer: $.jqplot.MeterGaugeRenderer,
      rendererOptions:renderOptions
    }
  });
  setInterval(updateMeter,30);

});
```

To create a new chart using the jqPlot library, we always call the `$.jqplot` function. The first parameter of the function is the `div` layer, which is where our work will live. The second parameter is a two-dimensional array containing the data of the chart (kind of looks odd for this example as it expects a 2D array and as our sample only includes one data entry at a time, we need to wrap it in two arrays). The third parameter defines the used renderer and `rendererOptions` (that we created earlier).

There's more...

Let's explore a few more functions.

Creating the updateMeter function

The `updateMeter` function gets called every 30 milliseconds. What we need to do is start by clearing our art every time that it is called:

```
meter.destroy();
```

This will clear everything related to our meter so we can recreate it.

If we are still in the intro part of our application where we want our speed to go up to the goal speed, we need to update our `meterValue` by `1`.

```
if(isStarting && meterValue<startingSpeed){
    ++meterValue;
}
```

If we are already passed this state and want our meter to go up and down randomly, making it look like variations in driving speed, we'll use the following code snippet:

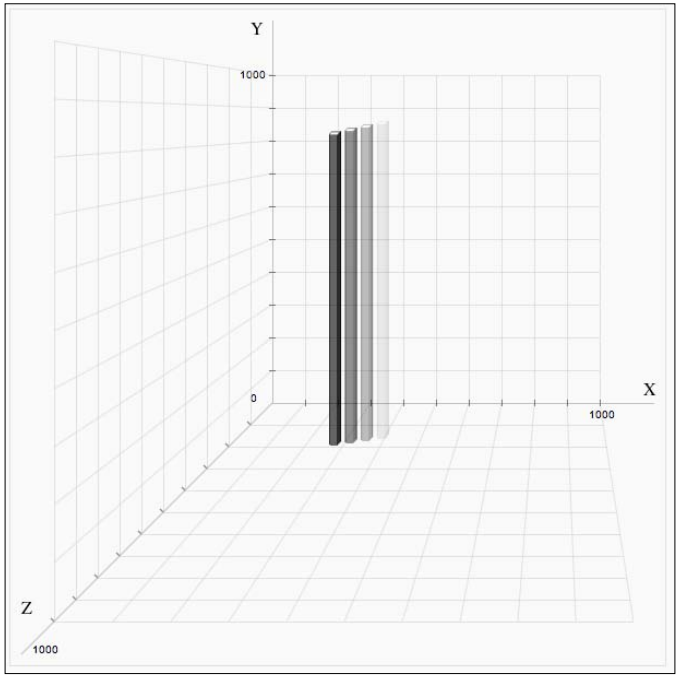
```
}else{
    meterValue += 1- Math.random()*2;
    meterValue = Math.max(0,Math.min(meterValue,120)); //keep our
    value in range no mater what
}
```

We are randomly adding a value between `-1` and `1` to our meter value. A correction to our result can be achieved by keeping our value not lower than `0` and not higher than `120`, followed by redrawing our meter with our new `meterValue` value.

Creating an animated 3D chart (canvas3DGraph)

This recipe is real fun. It's based on the source files of Dragan Bajcic. It's not a full library of charts, but it's a great inspirational chart that can be modified and used to create your own 3D data visualizations.

Although our source files in our attached sample are modified from the original source (mainly `canvas3DGraph.js`), to get the original source for the open source projects used in this book, please visit our centralized list at `http://blog.everythingfla.com/?p=339`.



Getting ready

If you want to follow our updates, download the original source files from the provided link or review the changes that we make to Dragan's source files.

How to do it...

Let's jump right in as we have a lot of work to do:

1. Create the HTML file:

```
<!DOCTYPE html>
<html>
  <head>
    <title>canvas3DGraph.js</title>
    <meta charset="utf-8" />
    <link rel="stylesheet"
    href="./external/dragan/canvas3DGraph.css">
    <script src="./external/dragan/
    canvas3DGraph.js"></script>
    <script src="./07.02.3d.js"></script>

  </head>
  <body style="background:#fafafa">

    <div id="g-holder">
      <div id="canvasDiv">
        <canvas id="graph" width="600" height="600" >
        </canvas>
        <div id="gInfo"></div>
      </div>

    </div>
  </body>
</html>
```

2. Create the CSS file `canvas3DGraph.css`:

```
#g-holder {
    height:620px;
    position:relative;
}

#canvasDiv{
    border:solid 1px #e1e1e1;
    width:600px;
    height:600px;
    position:absolute;
```

```
    top:0px; left:0px;
    z-index:10;
}
#x-label{
    position:absolute;
    z-index:2;
    top:340px;
    left:580px;
}

#y-label{
    position:absolute;
    z-index:2;
    top:10px;
    left:220px;
}

#z-label{
    position:absolute;
    z-index:2;
    top:540px;
    left:10px;
}

#gInfo div.gText{
    position:absolute;
    z-index:-1;
    font:normal 10px Arial;
}
```

3. Now it's time to move into the JavaScript file.
4. Let's add a few helper variables:

```
var gData = [];

var curIndex=0;
var trailCount = 5;
var g;
var trailingArray=[];
```

5. We need to create our chart when the document is ready:

```
window.onload=function(){
  //Initialize Graph
  g = new canvasGraph('graph');
  g.barStyle = {cap:'rgba(255,255,255,1)',main:
  'rgba(0,0,0,0.7)', shadow:'rgba(0,0,0,1)',
  outline:'rgba(0,0,0,0.7)',formater:styleFormater};
```

```
    for(i=0;i<100;i++){
      gData[i] = {x:(Math.cos((i/10)) * 400 + 400),
      y:(1000-(i*9.2)), z:(i*10)};
    }

plotBar();
setInterval(plotBar,40);


}
```

6. Create the `plotBar` function:

```
function plotBar(){
  trailingArray.push(gData[curIndex]);

  if(trailingArray.length>=5) trailingArray.shift();

  g.drawGraph(trailingArray);//trailingArray);
  curIndex++
  if(curIndex>=gData.length) curIndex=0;
}
```

7. Create the formatter function `styleFormatter`:

```
function styleFormatter(styleColor,index,total){
  var clrs = styleColor.split(",");
  var alpha = parseFloat(clrs[3].split(")"));
  alpha *= index/total+.1;
  clrs[3] = alpha+")";
  return clrs.join(",");
}
```

Assuming that you are using our modified, open source JavaScript file, you should now see your chart animated. (In the *There's more...* section in this recipe, we will look deeper into the changes and why we made them.)

How it works...

Let's first look at our code in the way that we interact with the JavaScript library. After that we will dig deeper into the inner workings of this library.

```
var gData = [];
var trailingArray=[];
var trailCount = 5;
var curIndex=0;
```

The `gData` array will store all the possible points in the 3D space. A 3D bar will be created with these points (the points are the 3D points x, y, and z values that will be put into this array as objects). The `trailingArray` array will store the current bar elements in the view. The `trailCount` variable will define how many bars can be seen at the same time, and our current index (`curIndex`) will keep track of our latest addition into the chart.

When the window loads we create our graph element:

```
window.onload=function(){
  //Initialise Graph
  g = new canvasGraph('graph');
  g.barStyle = {cap:'rgba(255,255,255,1)',main:'rgba(0,0,0,0.7)',
  shadow:'rgba(0,0,0,1)',outline:'rgba(0,0,0,0.7)',
  formatter:styleFormatter};
  for(i=0;i<100;i++){
    gData[i] = {x:(Math.cos((i/10)) * 400 + 400), y:(1000-
    (i*9.2)), z:(i*10)};
  }

  plotBar();
  setInterval(plotBar,40);


}
```

After creating our graph, we update the `barStyle` property to reflect the colors that we want to use on our bar. In addition to this, we are sending a formatter function as we want to treat each bar separately (visually treat them differently). We then create our data feed—in our case it's a traveling `Math.cos` in our inner space. Feel free to play around with all the data points; it creates some really amazing content. In a real-life application, you would want to use live or real data. To ensure that our data will be stacked from back to front, we would need to sort our data so that the z value that is in the back would be rendered first. In our case sorting isn't needed as our loop is creating an order of z indexes that grow in order, so the array is already organized.

There's more...

Next we call `plotBar` and repeat the action every 40 milliseconds.

The logic behind plotBar

Let's review the logic within the `plotBar` function. This is the really cool part of our application, where we update the data feed to create an animation. We start by adding the current index element into the `trailingArray` array:

```
trailingArray.push(gData[curIndex]);
```

If our array length is `5` or more, we need to get rid of the first element in the array:

```
if(trailingArray.length>=5) trailingArray.shift();
```

We then draw our chart and push the value of `curIndex` up by one. If our `curIndex` is greater than our array elements, we reset it to `0`.

```
g.drawGraph(trailingArray);//trailingArray);
curIndex++
if(curIndex>=gData.length) curIndex=0;
```

The logic behind styleFormatter

Our formatter function is called each time a bar is drawn to calculate the color to be used. It will get the index of the bar and the total length of the data feed in the chart being processed. In our example, we are only changing the `alpha` value of the bars based on their position. (The greater the number, the closer we are to the last entered data source.) In this way, we create our fade-out effect:

```
function styleFormatter(styleColor,index,total){
  var clrs = styleColor.split(",");
  var alpha = parseFloat(clrs[3].split(")"));
  alpha *= index/total+.1;
  clrs[3] = alpha+")";
  return clrs.join(",");
}
```

There is actually much more to this sample. Without going too deep into the code itself, I want to outline the changes.

To control the colors of our bars, line 66 of the third-party package has to be changed. As such, I introduced `this.barStyle` and replaced all the references of the hardcoded values during the creation of the bar elements (and set some default values):

```
this.barStyle = {cap:'rgba(255,255,255,1)',main:'rg
ba(189,189,243,0.7)', shadow:'rgba(77,77,180,0.7)',outline:'rgba(0,0,0
,0.7)',formatter:null};
```

I've created a style generator for our bars. This was done to help us redirect the logic between an external formatter and an internal style:

```
canvasGraph.prototype.getBarStyle= function(baseStyle,index,total){
  return this.barStyle.formatter?
  this.barStyle.formatter(baseStyle,index,total):baseStyle;
}
```

We have created a clear function to delete all the visuals from the graph so we can re-render the data each time we call it:

```
canvasGraph.prototype.getBarStyle= function(baseStyle,index,total){
  return this.barStyle.formatter?
  this.barStyle.formatter(baseStyle,index,total):baseStyle;
}
```

We moved the logic of drawing the chart to the `drawGraph` function, so I can delete the chart at the same time, making it easier for it to refresh all the data each time:

```
canvasGraph.prototype.drawGraph=function(gData){
  //moved this to the drawGraph so i can clear each time its
  called.
  this.clearCanvas();
  // Draw XYZ AXIS
  this.drawAxis();
  this.drawInfo();

  var len = gData.length;

  for(i=0;i<len;i++){
    this.drawBar(gData[i].x,gData[i].y,gData[i].z,i,len);
  }
}
```

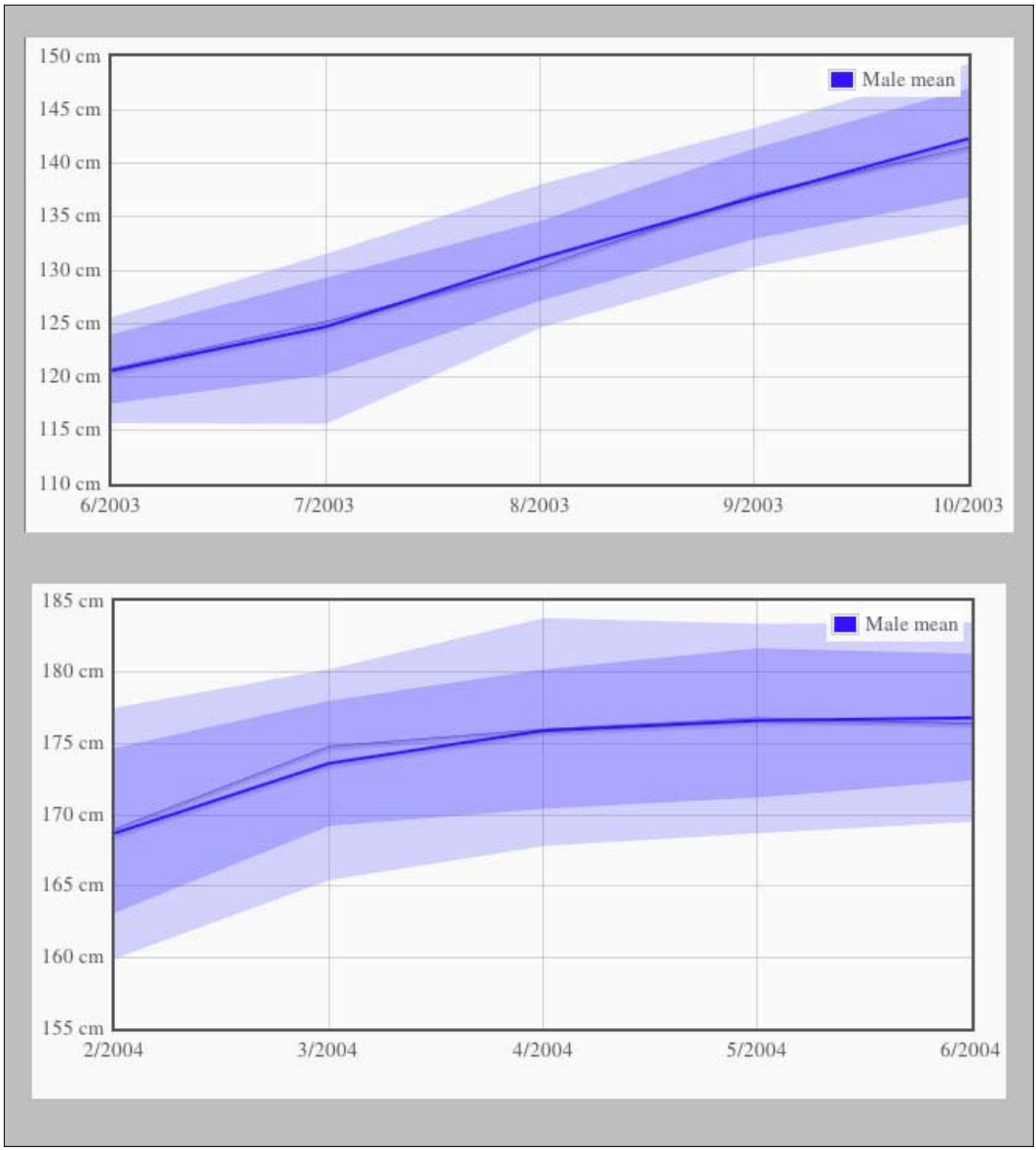
The current index and length information now travel through `drawBar` until it gets to the formatter function.

Last but not least, I've deleted the drawing of the chart from the constructor, so our chart will be more conducive for our animation idea.

Charting over time (flotJS)

One of the more impressive features of this library is the ease with which one can update the chart information. It's very easy to see from the first moment when you review this library and its samples that the author loves math and loves charting. My favorite feature is the way the chart can update its x ranges dynamically based on the input added into it.

My second favorite feature is how easy it is to update the chart text info by using a `tickFormater` method:



Getting ready

To get the latest builds of the `flotJS` library, please visit our link hub at `http://blog.everythingfla.com/?p=339` for charting open source libraries or download our book's source files where we include the latest build as of publication at `http://02geek.com/books/html5-graphics-and-data-visualization-cookbook.htm`.

How to do it...

Let's create our HTML and JavaScript files:

1. Create an HTML file:

```
<!DOCTYPE html>
<html>
  <head>
    <title>flot</title>
    <meta charset="utf-8" />
    <script src="http://ajax.googleapis.com/
    ajax/libs/jquery/1.7.2/jquery.min.js"></script>
    <script src="./external/flot/jquery.flot.js">
    </script>
    <script src="./external/flot/
    jquery.flot.fillbetween.js">
    </script>

    <script src="./07.03.flot.js"></script>

  </head>
  <body style="background:#fafafa">

    <div id="placeholder"
    style="width:600px;height:300px;"></div>
  </body>
</html>
```

2. Create a new JavaScript file (`07.03.flot.js`) and then create our data source:

```
var males = {

//…
//please grab from source files its a long list of numbers
};Create helper variables:
var VIEW_LENGTH = 5;
var index=0;
var plot;

var formattingData = {
```

```
    xaxis: { tickDecimals: 0, tickFormatter: function (v) {
    return v%12 + "/" + (2009+Math.floor(v/12)); } },
    yaxis: { tickFormatter: function (v) { return v + "
    cm"; } }
};
```

3. Create a `ready` event and trigger `updateChart`:

```
$(document).ready(updateChart);
```

4. Create `updateChart`:

```
function updateChart() {
  plot = $.plot($("#placeholder"), getData(),
  formattingData);

  if(index+5<males['mean'].length){
    setTimeout(updateChart,500);
  }
}
```

5. Create `getData`:

```
function getData(){
  var endIndex = index+5>=males.length?males.length-
  1:index+5;
  console.log(index,endIndex);
  var dataset = [
    { label: 'Male mean', data:
    males['mean'].slice(index,endIndex),
    lines: { show: true }, color: "rgb(50,50,255)" },
    { id: 'm15%', data:
    males['15%'].slice(index,endIndex),
    lines: { show: true, lineWidth: 0,
    fill: false }, color: "rgb(50,50,255)" },
    { id: 'm25%', data:
    males['25%'].slice(index,endIndex),
    lines: { show: true, lineWidth: 0, fill: 0.2 },
    color: "rgb(50,50,255)", fillBetween: 'm15%' },
    { id: 'm50%', data:
    males['50%'].slice(index,endIndex),
    lines: { show: true, lineWidth: 0.5, fill: 0.4,
    shadowSize: 0 }, color: "rgb(50,50,255)",
    fillBetween: 'm25%' },
    { id: 'm75%', data:
    males['75%'].slice(index,endIndex),
    lines: { show: true, lineWidth: 0, fill: 0.4 },
    color: "rgb(50,50,255)", fillBetween: 'm50%' },
    { id: 'm85%', data:
    males['85%'].slice(index,endIndex),
    lines: { show: true, lineWidth: 0, fill: 0.2 },
    color: "rgb(50,50,255)", fillBetween: 'm75%' }
```

```
    ];

    index++;
    return dataset;
  }
```

Now if you run the chart in your browser, you will see 6 months at a time, and at every half of a second, the chart will be updated by pushing the chart one month forward until the end of data source is reached.

How it works...

`flotJS` has a built-in logic to reset itself when its redrawn, and that's part of our magic. Our data source has been borrowed from one of the `flotJS` samples. We are actually using the data to represent a fictional situation. Originally this data was representing the average weight of people based on their age, broken down into percentiles. But our point in this example is not to showcase the data but instead show ways of visualizing the data. So in our case, we had to treat the data by keeping the percentiles as they are intended to be, but use the inner data to showcase the average over years instead of over ages, as follows:

```
{'15%': [[yearID, value], [yearID, value]...
```

The `yearID` values range from `2` through `19`. We want to showcase this information as if we started our data picking from 2006. Each `yearId` will represent a month (19 would be 1.5 years after 2006, instead of the age 19 as the data actually represents).

So let's start breaking it down. Now that we know how we are going to treat our dataset, we want to limit the number of months that we can see at any given time. As such we will add two helper parameters, one of which will keep track of our current index and the other will track the maximum number of visible elements at any given time:

```
var VIEW_LENGTH = 5;
var index=0;
```

We will create a global variable for our Flot graph and create a formatter to help us format the data that will be sent in.

```
var plot;
var formattingData = {
  xaxis: { tickDecimals: 0, tickFormatter: function (v) { return
  v%12 + "/" + (2003+Math.floor(v/12)); } },
  yaxis: { tickFormatter: function (v) { return v + " cm"; } }
};
```

Note that `tickFormater` enables us to modify the way our tick will look in the chart. In the case of the x axis, the goal is to showcase the current date `2/2012...`, and in the y axis, we want to add `cm` to the numbers that will be printed out on the screen.

There's more...

There are still two more things to cover—the `getData` function and the `updateChart` function.

The GetData function

In `flotJS` every data point has an ID. In our case, we want to showcase six related content types. Play around with the parameters to see how they change the way the view is rendered. Before we send the created array back, we update the index ID by one, so the next time that the function is called it will send the next range.

One more thing we need to note is the actual data range. As we are not sending the full data range (but a maximum of `5`), we need to validate that there are atleast five items after the index, and if not we will return the last element of the array, ensuring that we never slice more than the actual length:

```
var endIndex = index+5>=males.length?males.length-1:index+5;
```

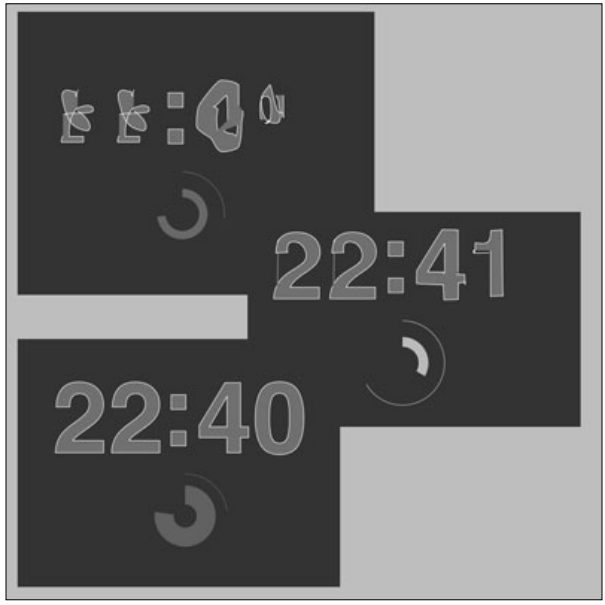
The UpdateChart function

This part is probably the simplest one. The same code is used for the first render and all the following renders. If the dataset is valid, we create a timeout and call this function again until the animation completes.

Building a clock with RaphaelJS

Hands down this is my favorite sample in this chapter. It's based on a mix of two samples on Raphael's website (I strongly encourage you to explore it). Although `Raphael` isn't a graphing library, it's a really powerful animation and drawing library that is really worth playing with.

In this recipe, we will create a clock that is creative (I think). I planned to play with this library for a day or two, and ended up playing with it all weekend as I was just having so much fun. I ended up with a digit morphing clock (based on a sample that Raphael created on his site for letter morphing) and incorporated some arcing into it based on the polar clock example on his site. Let's see it in action:



Getting ready

As always in this chapter, you need the original library of Raphael. I've added it into our project. So just download the files and let's get rolling.

To grab the original library, visit our external source files hub for this chapter at `http://blog.everythingfla.com/?p=339`.

How to do it...

Let's build up our application:

1. Create the HTML file:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Raphael</title>
    <meta charset="utf-8" />
    <script src="http://ajax.googleapis.com/ajax/
    libs/jquery/1.7.2/jquery.min.js"></script>
    <script src="./external/raphael/raphael-
    min.js"></script>
    <script src="./07.04.raphael.js"></script>
    <style>
```

```
        body {
          background: #333;
          color: #fff;
          font: 300 100.1% "Helvetica Neue", Helvetica,
          "Arial Unicode MS", Arial, sans-serif;
        }
        #holder {
          height: 600px;
          margin: -300px 0 0 -300px;
          width: 600px;
          left: 50%;
          position: absolute;
          top: 50%;
        }
      </style>

    </head>
    <body>

    <div id="holder"></div>
    </body>
</html>
```

2. Now it's time to move into the JavaScript file `07.04.raphael.js`. Copy the path parameters to draw the digits `0` through `9` and the `:` sign into an object called `helveticaForClock`. It's really just a long list of numbers, so please copy them from our downloadable source files:

```
var helveticaForClock = {…};
```

3. We are going to create an `onload` listener and put all of our code into it, to match it up with the style of code on Raphael's samples:

```
window.onload = function () {
  //the rest of the code will be put in here from step 3
  and on
};
```

4. Create a new `Raphael` object with a 600 x 600 size:

```
var r = Raphael("holder", 600, 600);
```

5. Now we need to use a helper function to figure out the path to an arc. For that we are going to create an `arc` function as an extra attribute for our newly created `Raphael` object:

```
r.customAttributes.arc = function (per,isClock) {
  var R = this.props.r,
  baseX = this.props.x,
  baseY = this.props.y;
```

```
    var degree = 360 *per;
    if(isClock) degree = 360-degree;

    var a = (90 - degree) * Math.PI / 180,
    x = baseX + R * Math.cos(a),
    y = baseY - R * Math.sin(a),
    path;

    if (per==1) {
      path = [["M", baseX, baseY - R], ["A", R, R, 0, 1, 1,
      baseX, baseY - R]];
    } else {
      path = [["M", baseX, baseY - R], ["A", R, R, 0,
      +(degree > 180), 1, x, y]];
    }

    var alpha=1;

    if(per<.1 || per>.9)
      alpha = 0;
    else
      alpha = 1;


    return {path: path,stroke: 'rgba(255,255,255,'+(1-
    per)+')'};
  };
```

6. Create our drawing of the hours of the clock (00:00):

```
var transPath;

var aTrans = ['T400,100','T320,100','T195,100','T115,100'];
var base0 = helveticaForClock[0];
var aLast = [0,0,0,0];
var aDigits = [];

var digit;
for(i=0; i<aLast.length; i++){
  digit = r.path("M0,0L0,0z").attr({fill: "#fff", stroke:
  "#fff", "fill-opacity": .3, "stroke-width": 1, "stroke-
  linecap": "round", translation: "100 100"});

  transPath = Raphael.
  transformPath(helveticaForClock[aLast[i]], aTrans[i]);
  digit.attr({path:transPath});
  aDigits.push(digit);
}
```

```
var dDot = r.path("M0,0L0,0z").attr({fill: "#fff", stroke: "#fff",
"fill-opacity": .3, "stroke-width": 1, "stroke-linecap": "round",
translation: "100 100"});
transPath = Raphael.transformPath(helveticaForClock[':'],
'T280,90');
dDot.attr({path:transPath});
```

7. Now it's time to create our art for our `seconds` animation:

```
var time;
var sec = r.path();
sec.props = {r:30,x:300,y:300}; //new mandatory params

var sec2 = r.path();
sec2.props = {r:60,x:300,y:300};

animateSeconds();
animateStrokeWidth(sec,10,60,1000*60);
```

8. Create the `animateSeconds` recursive function:

```
function animateSeconds(){ //will run forever
  time = new Date();

  sec.attr({arc: [1]});
  sec.animate({arc: [0]}, 1000, "=",animateSeconds);
  sec2.attr({arc: [1,true]});
  sec2.animate({arc: [0,true]}, 999, "=");

  var newDigits = [time.getMinutes()%10,
  parseInt(time.getMinutes()/10),
  time.getHours()%10,
  parseInt(time.getHours()/10)    ];
  var path;
  var transPath;
  for(var i=0; i<aLast.length; i++){
    if(aLast[i]!=newDigits[i]){
      path = aDigits[i];
      aLast[i] = newDigits[i];
      transPath = Raphael.transformPath
      (helveticaForClock[newDigits[i]], aTrans[i]);
      path.animate({path:transPath}, 500);
    }
  }

}
```

9. Create the `animateStrokeWidth` function:

```
function animateStrokeWidth(that,startWidth,endWidth,time){
  that.attr({'stroke-width':startWidth});
  that.animate({'stroke-width':endWidth},time,function(){
    animateStrokeWidth(that,startWidth,endWidth,time); //repeat
forever
  });
}
```

If you run the application now, you will see the outcome of my day of play with Raphael's library.

How it works...

There are a lot of elements to this project. Let's start focusing on the arc animation. Note that one of the elements that we are using in our code is when we are creating our new paths (we create two of them). We are adding some hardcoded parameters that will be used later when we draw the arcs in the `arc` method:

```
var sec = r.path();sec.props = {r:30,x:300,y:300}; //new mandatory
params

var sec2 = r.path();sec2.props = {r:60,x:300,y:300};
```

We are doing that to avoid sending these three properties into the arc each time, and to enable us to pick a radius and stick with it without it being integrated or hardcoded into the animations. We based our `arc` method on the `arc` method used for the polar clock in Raphael's examples, but changed it so the values can be positive or negative (making it easier to animate back and forth).

The `arc` method is then used to draw our arc when we are animating it inside the `animateSeconds` function:

```
sec.attr({arc: [1]});
sec.animate({arc: [0]}, 1000, "=",animateSeconds);
sec2.attr({arc: [1,true]});
sec2.animate({arc: [0,true]}, 999, "=");
```

The `attr` method will reset our `arc` attribute so that we can reanimate it.

By the way, note that in `animateStrokeWidth` we are animating the width of our stroke for 60 seconds from its lowest value to its highest value.

There's more...

Did you really think we are done? I know you didn't. Let's take a look at a few other critical steps.

Animating paths

One of the cooler things in this library is the capability to animate paths. If you have ever worked with Adobe Flash Shape Tweens, this will look very familiar—hands down, this is just really cool.

The idea is very simple. We have an object with a lot of path points. They create a shape together if we draw the line information through them. We have borrowed a list that Raphael created so we don't need to start from scratch, and literally all that we are changing in it is that we don't want our elements to be drawn in their current path. All we need to do is transform their location using the internal `Raphael.transformPath` method:

```
transPath = Raphael.transformPath(helveticaForClock[0], 'T400,100');
```

In other words, we are grabbing the path information for the digit 0 and then we are transforming, moving it 400 pixels to the right and 100 pixels down.

In our source code, it looks like we are executing the function in a loop (which is a bit more complicated but condensed):

```
for(i=0; i<aLast.length; i++){
  digit = r.path("M0,0L0,0z").attr({fill: "#fff", stroke: "#fff",
  "fill-opacity": .3, "stroke-width": 1, "stroke-linecap":
  "round", translation: "100 100"});

  transPath = Raphael.transformPath(helveticaForClock[aLast[i]],
  aTrans[i]);
  digit.attr({path:transPath});
  aDigits.push(digit);
}
```

We are basically looping through the `aLast` array (the list of digits that we want to create) and creating a new digit for each element. We then figure out the position of the digit based on the transforming information that is located in the `aTrans` array and then we draw it out by adding a new path into the attributes. Last but not least, we are saving our digit into our `aDigits` array that is to be used when we re-render the element later.

Each time the `animateSeconds` function gets called (once every second), we figure out if a digit has changed, and if it has then we are ready to update its information:

```
var newDigits = [time.getMinutes()%10,
  parseInt(time.getMinutes()/10),
  time.getHours()%10,
  parseInt(time.getHours()/10)];
var path;
var transPath;
  for(var i=0; i<aLast.length; i++){
    if(aLast[i]!=newDigits[i]){
    path = aDigits[i];
    aLast[i] = newDigits[i];
    transPath = Raphael.transformPath
    (helveticaForClock[newDigits[i]], aTrans[i]);
    path.animate({path:transPath}, 500);
  }
}
```

We start by gathering the current time `HH:MM` into an array (`[H,H,M,M]`) followed by looking to see if our digits have changed. If they have changed, we grab the new data needed from our `helveticaForClock` function and animate it in our new path information for our digit (path).

That covers the most important factors for following this recipe.

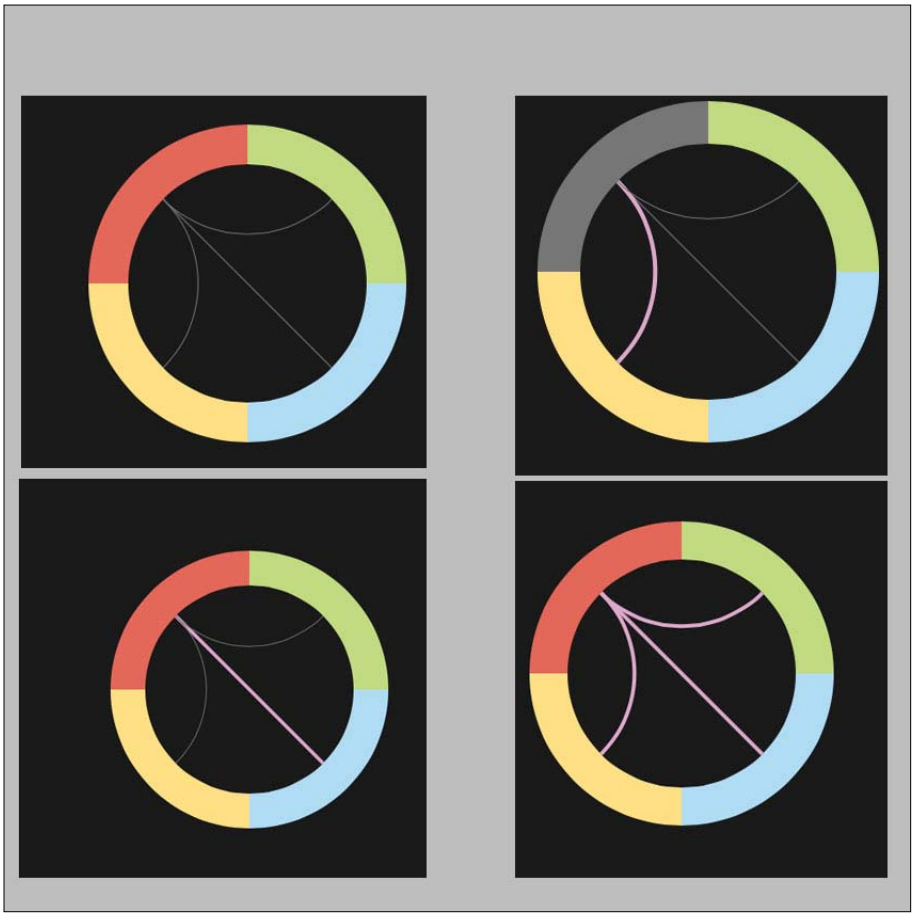
Making a sunburst chart with InfoVis

Another really cool library is `InfoVis`. If I had to categorize the library, I would say it's about connections. When you review the rich samples provided by Nicolas Garcia Belmonte, you will find a lot of relational datatypes that are very unique.

This library is distributed freely through Sencha legal owners. (The copyright is easy to follow, but please review the notes for this and any open source project that you encounter.)

We will start with one of his base samples—the sunburst example from the source files. I've made a few changes to give it a new personality. The basic idea of a sunburst chart is to showcase relationships between nodes. While a tree is an ordered parent-child relationship, the relationships in a sunbust chart are bidirectional. A node can have a relationship with any other node, and it can be a two-way or one-way relationship. A dataset that is perfect for this is the example of the total exports of a country—lines from one country to all the other countries that get exports from it.

We will keep it relatively simple by having only four elements (Ben, Packt Publishing, 02geek, and Nicolas the creator of InfoVis). I have a one-way relationship with each of them: as the owner of `02geek.com`, as a writer for Packt Publishing, and a user of InfoVis. While that is true about me, not all the others have a real in-depth relationship with me. Some of them have a relationship back with me, such as 02geek and Packt Publishing, while Nicolas for this example is a stranger that I've never interacted with. This can be depicted in a sunburst chart in the following way:



Getting ready

As always you will need the source files, you can either download our sample files or get the latest release by visiting our aggregated list at `http://blog.everythingfla.com/?p=339`.

How to do it...

Let's create some HTML and JavaScript magic:

1. Create an HTML file as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Sunberst - InfoVis</title>
    <meta charset="utf-8" />

    <style>
      #infovis {
        position:relative;
        width:600px;
        height:600px;
        margin:auto;
        overflow:hidden;
      }
    </style>

    <script  src="./external/jit/jit-yc.js"></script>
    <script src="./07.05.jit.js"></script>
  </head>

  <body onload="init();">
    <div id="infovis"></div>
  </body>
</html>
```

2. The rest of the code will be in `07.05.jit.js`. Create a base data source as follows:

```
var dataSource = [ {"id": "node0", "name": "","data": {"$type":
"none" },"adjacencies": []}]; //starting with invisible root
```

3. Let's create a function that will create the nodes needed for our chart system:

```
function createNode(id,name,wid,hei,clr){
  var obj = {id:id,name:name,data:{"$angularWidth":wid,
  "$height":hei,"$color":clr},adjacencies:[]};
  dataSource[0].adjacencies.push({"nodeTo": id,"data":
  {'$type': 'none'}});
  dataSource.push(obj);

  return obj;
}
```

4. To connect the dots, we will need to create a function that will create the relationships between the elements:

```
function relate(obj){
  for(var i=1; i<arguments.length; i++){
    obj.adjacencies.push({'nodeTo':arguments[i]});
  }
}
```

5. We want to be able to highlight the relationships. To do that we will need to have a way to rearrange the data and highlight the elements that we want highlighted:

```
function highlight(nodeid){
  var selectedIndex = 0;
  for(var i=1; i<dataSource.length; i++){
    if(nodeid!=     dataSource[i].id){
      for(var item in dataSource[i].adjacencies)
      delete dataSource[i].adjacencies[item].data;
    }else{
      selectedIndex = i;
      for(var item in dataSource[i].adjacencies)
      dataSource[i].adjacencies[item].data =  {"$color":
      "#ddaacc","$lineWidth": 4 };
      }

    }

    if(selectedIndex){ //move selected node to be first
    (so it will highlight everything)
    var node = dataSource.splice(selectedIndex,1)[0];
    dataSource.splice(1,0,node);
  }

}
```

6. Create an `init` function:

```
function init(){
/* or the remainder of the steps
all code showcased will be inside the init function  */
}
```

7. Let's start building up data sources and relationships:

```
function init(){
  var node =
  createNode('geek','02geek',100,40,"#B1DDF3");
  relate(node,'ben');
  node = createNode('packt','PacktBub',100,40,"#FFDE89");
```

```
  relate(node,'ben');
  node = createNode('ben','Ben',100,40,"#E3675C");
  relate(node,'geek','packt','nic');

  node = createNode('nic','Nicolas',100,40,"#C2D985");
  //no known relationships so far ;)
...
```

8. Create the actual sunburst and interact with the API (I've stripped it down to its bare bones; in the original samples it's much more detailed):

```
var sb = new $jit.Sunburst({
  injectInto: 'infovis', //id container
  Node: {
    overridable: true,
    type: 'multipie'
  },
  Edge: {
    overridable: true,
    type: 'hyperline',
    lineWidth: 1,
    color: '#777'
  },
  //Add animations when hovering and clicking nodes
  NodeStyles: {
    enable: true,
    type: 'Native',
    stylesClick: {
    'color': '#444444'
  },
  stylesHover: {
    'color': '#777777'
  },
    duration: 700
  },
  Events: {
    enable: true,
    type: 'Native',
    //List node connections onClick
    onClick: function(node, eventInfo, e){
      if (!node) return;

      highlight(node.id);
      sb.loadJSON(dataSource);
      sb.refresh()
    }
  },
  levelDistance: 120
});
```

9. Last but not least, we want to render our chart by providing its `dataSource` and refresh the render for the first time:

```
sb.loadJSON(dataSource);
sb.refresh();
```

That's it. If you run the application, you will find a chart that is clickable and fun, and just scratches the capabilities of this really cool data networking library.

How it works...

I'll avoid getting into the details of the actual API as that is fairly intuitive and has a really nice library of information and samples. So instead I will focus on the changes and enhancements that I've created in this application.

Before we do that we need to understand how the data structure of this chart works. Let's take a deeper look into how the data source object will look when filled with information:

```
{
        "id": "node0",
        "name": "",
        "data": {
          "$type": "none"
        },
        "adjacencies": [
            {"nodeTo": "node1","data": {'$type': 'none'}},
            {"nodeTo": "node2","data": {'$type': 'none'}},
            {"nodeTo": "node3","data": {'$type': 'none'}},
            {"nodeTo": "node4","data": {'$type': 'none'}}
                      ]
},

{
        "id": "node1",
        "name": "node 1",
        "data": {
          "$angularWidth": 300,
          "$color": "#B1DDF3",
          "$height": 40
        },
        "adjacencies": [
            {
              "nodeTo": "node3",
              "data": {
                "$color": "#ddaacc",
```

```
                "$lineWidth": 4
              }
            }
                    ]
},
```

There are a few important factors to note. The first is that there is a base parent that is the parent of all the parentless nodes. In our case it's a flat chart. The relationships that are really thrilling are between nodes that are at an equal level. As such the main parent has a relationship with all the nodes that are to follow. The children elements, such as `node1` in this case, could have relationships. They are listed out in an array called `adjacencies` that holds objects. The only mandatory parameter is the `nodeTo` property. It lets the application know the one-way relationship list. There are optional layout parameters that we will add later only when we want to highlight a line. So let's see how we can create this type of data dynamically with the help of a few functions.

The `createNode` function helps us keep our code clean by wrapping up the dirty steps together. Every new element that we add needs to be added into our array and is needed to update our main parent (that is always going to be in position `0` of our array of new elements):

```
function createNode(id,name,wid,hei,clr){
  var obj = {id:id,name:name,data:{"$angularWidth":wid,
  "$height":hei,"$color":clr},adjacencies:[]};
  dataSource[0].adjacencies.push({"nodeTo": id,"data":
  {'$type': 'none'}});
  dataSource.push(obj);

  return obj;
}
```

We return the object as we want to continue and build up the relationship with this object. As soon as we create a new object (in our `init` function), we call the `relate` function and send to it all the relationships that our element will have to it. The logic of the `relate` function looks more complicated that it actually is. The function uses a hidden or often ignored feature in JavaScript that enables developers to send an open-ended number of parameters into a function with the use of the `arguments` array that is created automatically within every function. We can get these parameters as an array named `arguments`:

```
function relate(obj){
  for(var i=1; i<arguments.length; i++){
    obj.adjacencies.push({'nodeTo':arguments[i]});
  }
}
```

The `arguments` array is built into every function and stores all the actual information that has been sent into the function. As the first parameter is our object, we need to skip the first parameter and then add the new relationships into the `adjacencies` array.

Our last data-related function is our `highlight` function. The `highlight` function expects one parameter `nodeID` (that we created in `createNode`). The goal of the `highlight` function is to travel through all the data elements and de-highlight all the relationships limited to the one selected element and its relationships.

```
function highlight(nodeid){
  var selectedIndex = 0;
  for(var i=1; i<dataSource.length; i++){
    if(nodeid!=  dataSource[i].id){
      for(var item in dataSource[i].adjacencies)
      delete dataSource[i].adjacencies[item].data;
    }else{
      selectedIndex = i;
      for(var item in dataSource[i].adjacencies)
      dataSource[i].adjacencies[item].data =  {"$color":
      "#ddaacc","$lineWidth": 4 };
    }

  }
}
```

If we don't have `highlight`, we want to confirm and remove all the instances of the data object within the adjacencies of the node, while if it is selected, we need to add that same object by setting it with its own color and a thicker line.

We are almost done with the data. But when running the application, you will find an issue if we stop here. The issue is within the way the chart system works. If a line was drawn it will not redraw it again. In practical terms, if we select "Ben" while `ben` isn't the first element in the list, then not all the relationships that "Ben" has with the others will be visible. To fix this issue, we would want to push the selected node to be the first element right after position `0` (main parent), so it will render the selected relationships first:

```
if(selectedIndex){
  var node = dataSource.splice(selectedIndex,1)[0];
  dataSource.splice(1,0,node);
}
```

There's more...

One more thing left is that we need to be able to refresh our content when the user clicks on an element. To accomplish this task, we will need to add an event parameter into the initializing parameter object of `jit.Sunburst`:

```
var sb = new $jit.Sunburst({
  injectInto: 'infovis', //id container
     ...
  Events: {
    enable: true,
    type: 'Native',
    //List node connections onClick
    onClick: function(node, eventInfo, e){
      if (!node) return;

      highlight(node.id);
      sb.loadJSON(dataSource);
        sb.refresh();
    }
  },
  levelDistance: 120
});
```

One more thing to note in this sample is the `levelDistance` property that controls how close/far you are to/from the rendered element (making it bigger or smaller).

Where is the copy?

There is still one more issue. We don't have any copy in our chart enabling us to know what is actually being clicked on. I've removed it from the original sample as I just didn't like the positioning of the text and couldn't figure out how to get it up right, so instead I came up with a workaround. You can directly draw into the canvas by directly interacting with it. The canvas element will always be called by the same ID as our project (in our case `infovis` followed by `-canvas`):

```
var can = document.getElementById("infovis-canvas");
  var context = can.getContext("2d");
…
```

I'll leave the rest for you to explore. The rest of the logic is easy to follow as I've stripped it down. So if you enjoy this project as well, please visit the InfoVis Toolkit site and play more with their interface options.