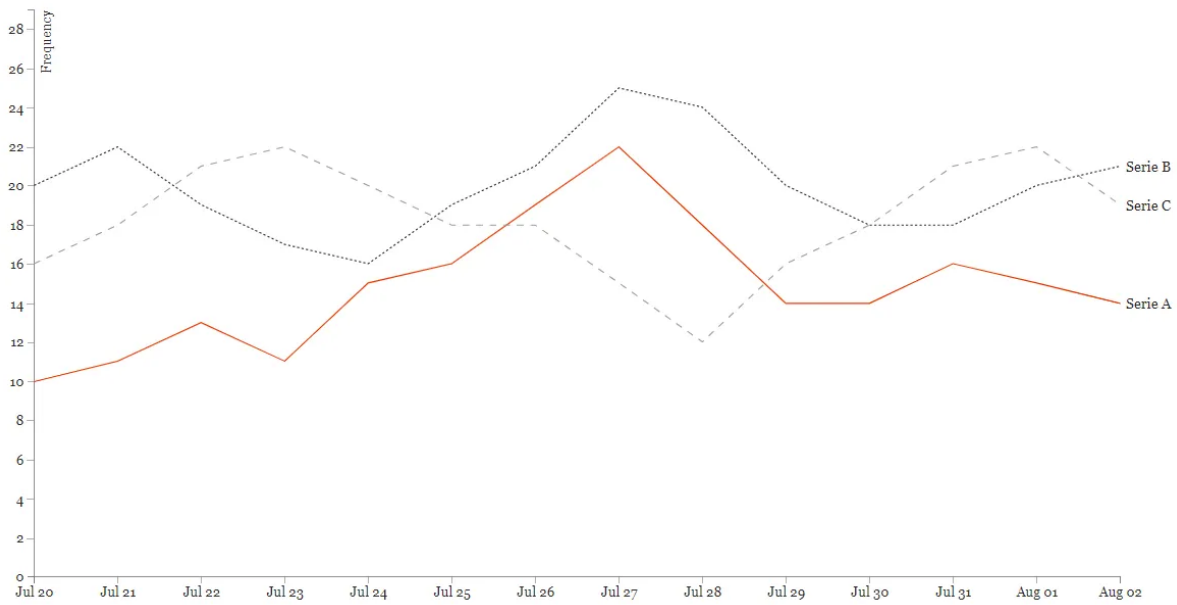
TUTORIAL: Making a Line Chart in D3.js v.5

Posted on 28th October 2019 by Eve the Analyst

The time has come to step up our game and create a line chart from scratch. And not just *any* line chart: a multi-series graph that can accommodate any number of lines. Besides handling multiple lines, we will work with time and linear scales, axes, and labels – or rather, have them work for us. There is plenty to do, so I suggest you fire off your D3 server and let's get cracking.



We will create this beauty! Fun!

Document Preparation

As the first step we need to prepare the data and the file structure for the visualisation. Create *line_chart.html*, *styles.css*, and *data.csv* in your project folder and

populate them with the following snippets to get us started.

Paste this to the *line_chart.html* file. The code defines the svg element for us so we can start drawing straight away. I've also created a basic structure upfront so it's easier to navigate the document as we work on its particular sections.

```
<!DOCTYPE html>
<head>
<meta charset="utf-8">
<title>Multi Line Chart</title>
<script type="text/javascript" src="https://d3js.org/d3.v5.min.js"></script>
<link rel="stylesheet" type="text/css" href="styles.css">
<style></style>
</head>
<body>
<div id="container" class="svg-container"></div>
<script>
//------------------------1. PREPARATION-------------------------//
//-----------------------------SVG-------------------------------//
const width = 960;
const height = 500;
const margin = 5;
const padding = 5;
const adj = 30;
// we are appending SVG first
const svg = d3.select("div#container").append("svg")
   .attr("preserveAspectRatio", "xMinYMin meet")
   .attr("viewBox", "-"
        + adj + " -"
        + adj + " "
        + (width + adj *3) + " "
        + (height + adj*3))
   .style("padding", padding)
   .style("margin", margin)
```

```
    .classed("svg-content", true);

//----------------------------DATA----------------------------//

//---------------------------SCALES---------------------------//

//----------------------------AXES----------------------------//

//----------------------------LINES---------------------------//

//-------------------------2. DRAWING-------------------------//

//----------------------------AXES----------------------------//

//----------------------------LINES---------------------------//


});
</script>
```

Leave the *styles.css* document empty for now. Paste the following rows to *data.csv*. The line chart will feature multiple series: corresponding to the columns A, B, and C.

```
date,A,B,C
20-Jul-2019,10,20,16
21-Jul-2019,11,22,18
22-Jul-2019,13,19,21
23-Jul-2019,11,17,22
24-Jul-2019,15,16,20
25-Jul-2019,16,19,18
26-Jul-2019,19,21,18
```

27-Jul-2019,22,25,15
28-Jul-2019,18,24,12
29-Jul-2019,14,20,16
30-Jul-2019,14,18,18
31-Jul-2019,16,18,21
01-Aug-2019,15,20,22
02-Aug-2019,14,21,19

Data Preparation

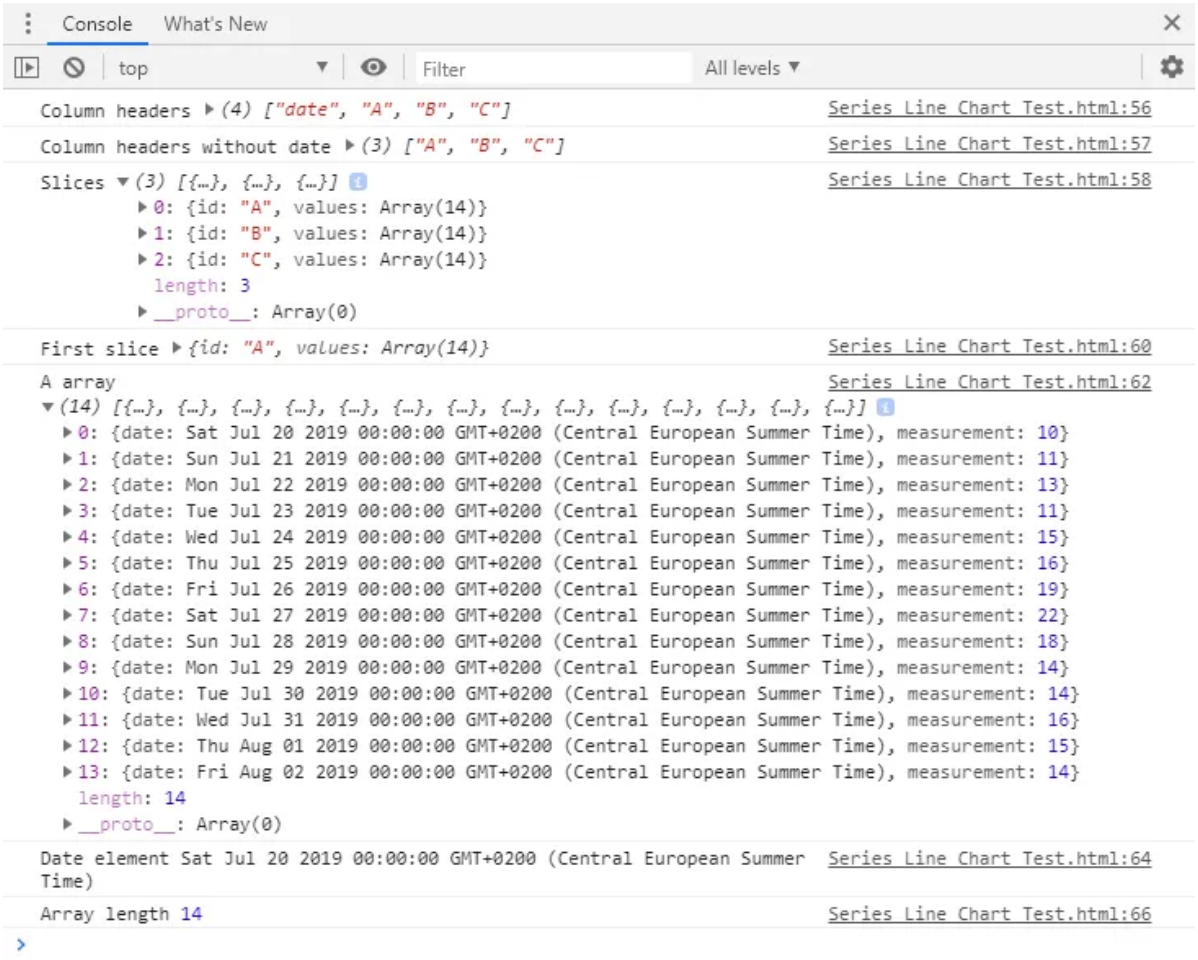
The first step – and a crucial step for the whole visualisation – is to correctly read the data. I have used a multi-series example for a reason: while it's pretty simple to plot a single path, handling multiple lines (especially an undefined number of them) requires a bit more of D3 finesse. Paste the following to the DATA section, reload the html document and review the console log in your browser:

```
const timeConv = d3.timeParse("%d-%b-%Y");
const dataset = d3.csv("data.csv");
dataset.then(function(data) {
  const slices = data.columns.slice(1).map(function(id) {
    return {
      id: id,
      values: data.map(function(d){
        return {
          date: timeConv(d.date),
          measurement: +d[id]
        };
      })
    };
  });

console.log("Column headers", data.columns);
```

```
console.log("Column headers without date", data.columns.slice(1));
// returns the sliced dataset
console.log("Slices",slices);
// returns the first slice
console.log("First slice",slices[0]);
// returns the array in the first slice
console.log("A array",slices[0].values);
// returns the date of the first row in the first slice
console.log("Date element",slices[0].values[0].date);
// returns the array's length
console.log("Array length",(slices[0].values).length);
```

Let's review the transformations called on our data set one by one:
– *data.columns* returns the csv headers
– *data.columns.slice(1)* returns the csv headers without the date column (the slice starts at column indexed at 1)
– *map()* calls a function on every element of the array (consisting of A, B, and C) – let's call each of those elements a '*slice*'
– *map()* assigns the column name as an *id* element to each *slice*
– then it assigns a *values* array to each *slice*
– note how the *values* element evokes a function. Here we map information from the original data set: the array will consist of 2 columns, *date* and *measurement*. The *date* is derived from the first column (and transformed to a date format), and the *measurement* is taken from the column corresponding with the *slice*'s *id*.
At the end of those transformations we get 3 arrays: A, B, and C, with 2 columns each: date and measurement. This is an amazingly flexible way to slice up a data set: regardless of how many columns it has! It's all done in those few rows. I've printed some information to the console to help you review the snippet.



Review the console log for more information

Scales Preparation

After the data has been read in we need to configure the scaling mechanism. This is done in order to print the chart in accordance to the svg's real estate. Scales transform the data input (our dates and values) to coordinates on the svg plane. Paste the following lines to the SCALES section.

```
const xScale = d3.scaleTime().range([0,width]);
const yScale = d3.scaleLinear().rangeRound([height, 0]);
xScale.domain(d3.extent(data, function(d){
   return timeConv(d.date)}));
yScale.domain([(0), d3.max(slices, function(c) {
   return d3.max(c.values, function(d) {
      return d.measurement + 4; });
```

```
        })
    ]);
```

We will plot the dates on the x axis and the values on the y axis. D3 provides a *scaleTime()* method for scaling dates, and a *scaleLinear()* method for continuous values. We first decide the scale's range: what the input values should be translated to. In this case we will stretch the data values from 0 to the svg's width, and the numeric values from the svg's height to 0. As the second step we specify the input data domain. A domain consists of all values between a specified minimum and maximum that a data set can take. Instead of manually looking up those values, we pass it through built in D3 functions:
– *d3.extent()* returns a minimum and maximum value of an array (in a natural order) – this will work perfectly on our date set
– *d3.max()* returns a maximum value of the array. Note how in this example we first extract a maximum value from each array to then select a maximum of all three. I also added 4 to the maximum value for purely subjective aesthetic reasons: I wanted to have some space above the graph.

The scales are now set up. If you don't have enough of scales and would like to see more examples, take a look at my previous tutorial.

A well configured scales enable us to start plotting values on the svg. Every time we evoke the data set, we just need to call an appropriate scale on it.

Axes

Enough chatting – let's draw something already! Axes are a good starting point: if plotted correctly, they will assure us that the data has been read as expected and that it scales as nicely as we have imagined.

Paste this to AXES under the Preparation section:

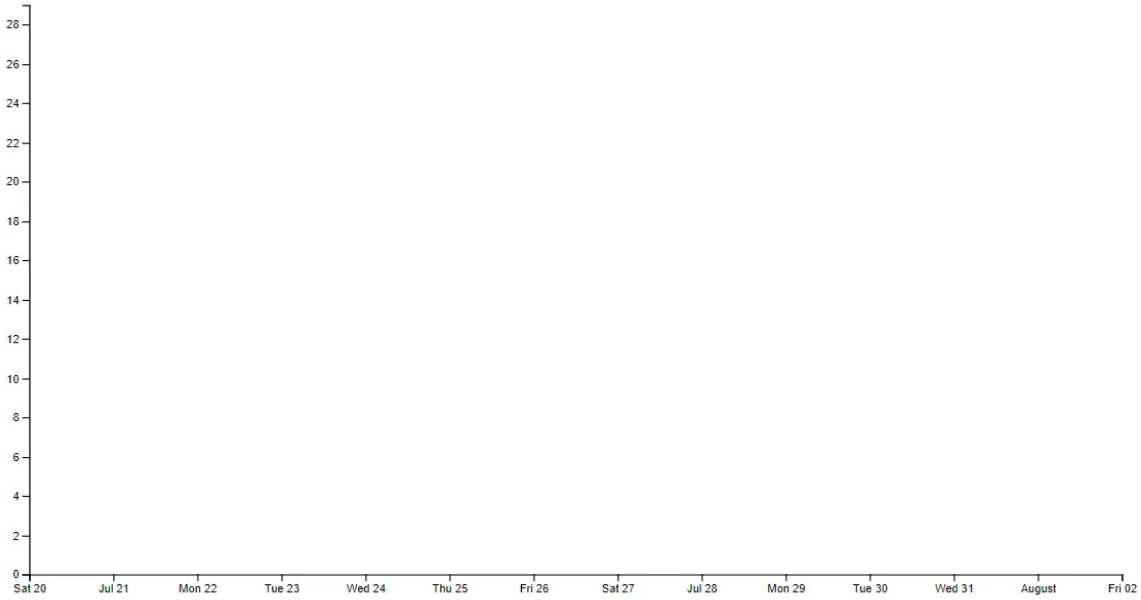
```
const yaxis = d3.axisLeft().scale(yScale);
const xaxis = d3.axisBottom().scale(xScale);
```

And this to AXES under the Drawing section:

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

svg.append("g")
    .attr("class", "axis")
    .call(yaxis);
```

And as simple as that we have plotted the x and y axes!



The axes are here!

Admittedly, the axes are not the most elegant in the world (there are some pretty axes out there) but they are here! There are some additional tweaks we can apply to make them more friendly for the reader.

Let's look at the x axis first: there is something funny going on with the dates. Reading from the left, we get 'Sat 20', 'Jul 21', 'Mon 22', and at some point we reach just 'August'. Seems like the months and days have come in an insubordinate mix of variations. We need to put and end to this freestyle, and by this I mean that we should decide what date format we'd like to print on the screen. The *d3.axis()* method allows us to adjust all sort of things for ticks – their number, interval between the points, display format, etc. Let's configure some of them for both axes.

Replace the axes definition in the Preparation section with the following snippet and refresh the visualisation:

```
const yaxis = d3.axisLeft()
    .ticks((slices[0].values).length)
    .scale(yScale);

const xaxis = d3.axisBottom()
    .ticks(d3.timeDay.every(1))
    .tickFormat(d3.timeFormat('%b %d'))
    .scale(xScale);
```

The above code specifies a set number of ticks for the y axis (14, or as many as there are array elements / csv rows). In the case of the x axis a tick will be displayed with a granularity of a day, every day. That's been achieved by setting the tick property to d3.timeDay.every(1). The format of displayed dates will show the day and the abbreviated month for each tick. After those changes we end up with somewhat improved axes:

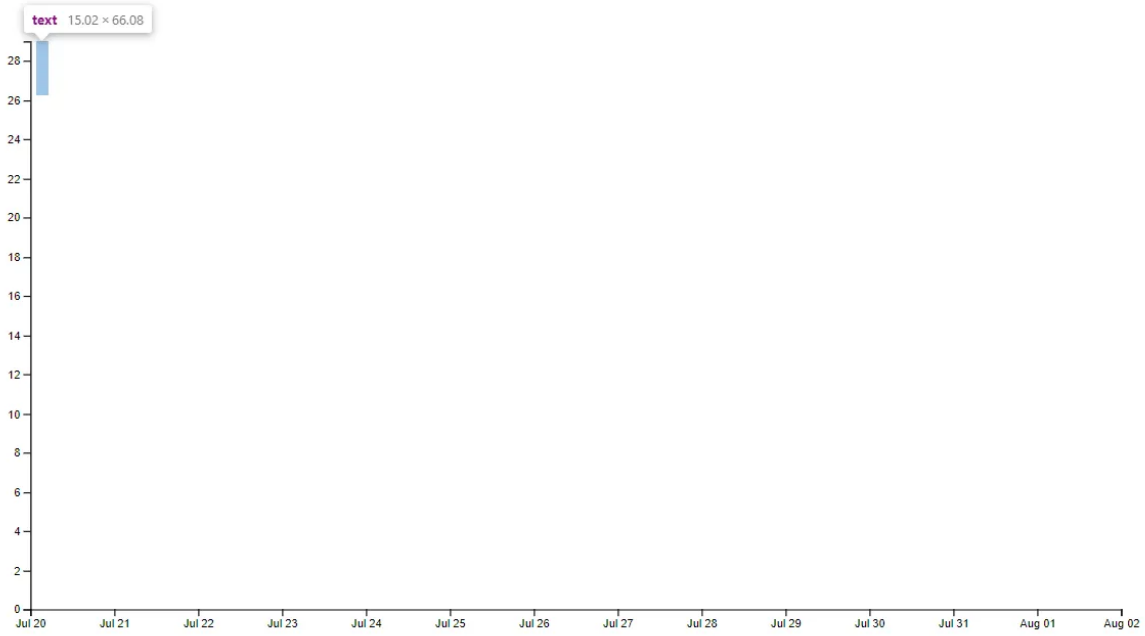
Everything is in order now

Disobedient dates are no longer a problem!

To make it even better (is it even possible!!!) we can add a label to the y axis to show what the values stand for. While the dates are self-explanatory, the numbers on their own carry no information. Add a label (call it whatever you like – I went with *Frequency*) by appending the following to the y axis drawing:

```
//this you had
svg.append("g")
    .attr("class", "axis")
    .call(yaxis)
//this you append
    .append("text")
    .attr("transform", "rotate(-90)")
    .attr("dy", ".75em")
    .attr("y", 6)
    .style("text-anchor", "end")
    .text("Frequency");
```

(There is no style set for the label so it won't show on the graph – but believe me and the Google Chrome developer tools, it is there)



The y axis label is invisible

Finally, let's improve the looks of the axes. By improve I mean: set the colours, widths, and rendering of every single element, and decide the font to use. Paste the following to the css file and feel free to make your own style decisions:

```
/* AXES */
/* ticks */
.axis line{
stroke: #706f6f;
stroke-width: 0.5;
shape-rendering: crispEdges;
}

/* axis contour */
.axis path {
stroke: #706f6f;
stroke-width: 0.7;
shape-rendering: crispEdges;
```

```
}

/* axis text */
.axis text {
fill: #2b2929;
font-family: Georgia;
font-size: 120%;
}
```

The ticks are controlled by the *.line* element of the axis, while the actual axis is set with the *.path* element. The axes look sharp (the author's humble opinion) and ready to welcome some data:

Without further ado, let's plot the chart!

Line Chart

Lines are essentially d3.paths() that connect a bunch of (x, y) coordinates on a 2D plane. To construct a line you need to tell it where to find its x and y coordinates and then append that to the svg. Paste the following snippets to the previously created placeholders and let's review the code together.

This should got to LINES bit under the Preparation section:

```
const line = d3.line()
    .x(function(d) { return xScale(d.date); })
    .y(function(d) { return yScale(d.measurement); });
```

In this snippet we called a line constructor, *d3.line()* that uses two accessors: *x* for values on the horizontal plane, and *y* for the vertical axis. Here we simply point to the most granular values of our array, date and measurement (this is not the time to worry about the nested csv structure). Once that's done, paste the following to LINES under the Drawing section:

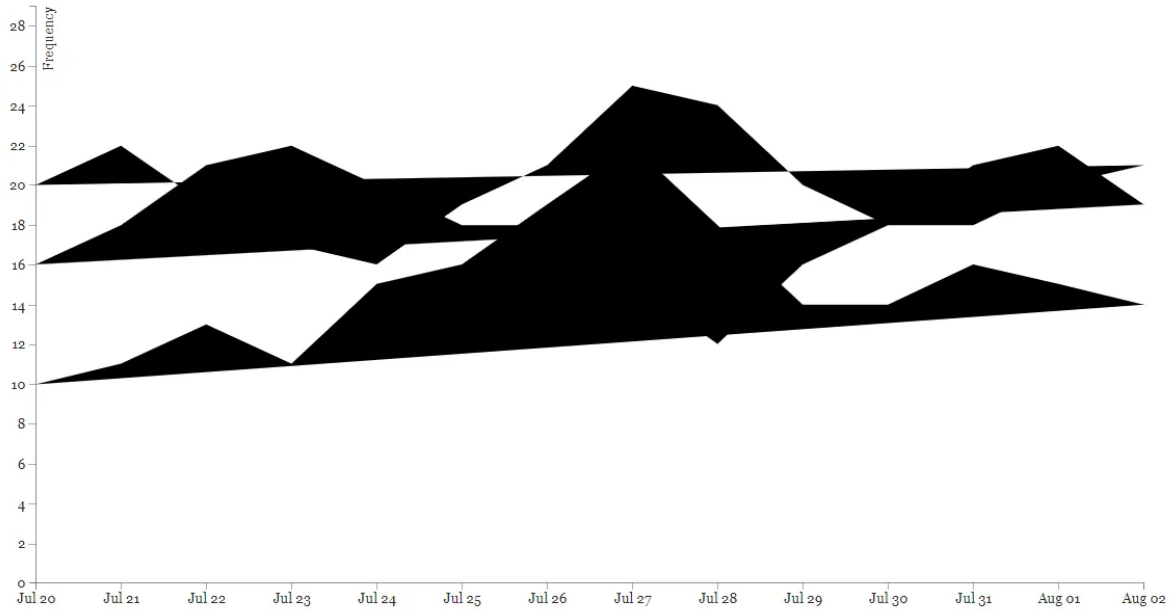
```
const lines = svg.selectAll("lines")
    .data(slices)
    .enter()
    .append("g");

    lines.append("path")
    .attr("d", function(d) { return line(d.values); });
```

This requires some explanation. The variable *lines* selects an unidentified number of lines from the svg – and immediately tells D3 there will be 3 lines by pointing to the *slices* set (lines A, B, and C). It then appends a g element to each of them: a grouping element that will make our life easier in the due course. The g element will collect everything that has to do with a particular chart series (aka a slice in the array): the line (represented above as a path), its data points that we will be able to hover over, and the series labels.

The first thing to append to *lines* (that are in fact 3 empty *g* containers) are the chart lines themselves. We call the *d3.line()* constructor on the data to draw a path. See how first we need to access the *values* under each slice. This is then passed to the constructor that pulls *dates* and *measurements* as required.

After the changes are saved, the visualisation gets updated to this:

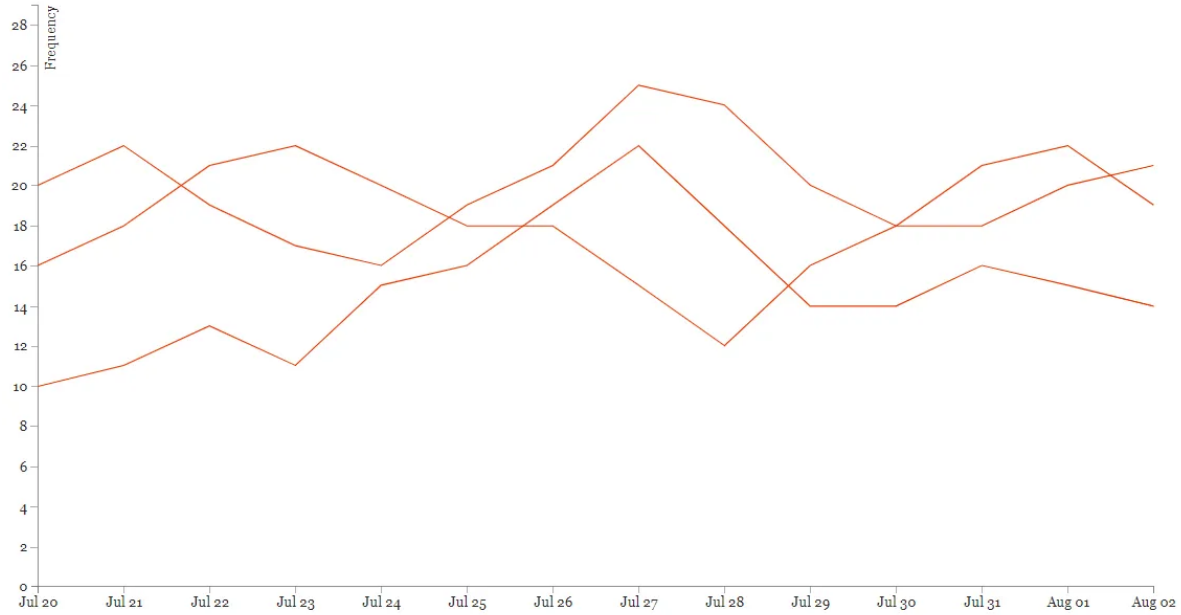


Instead of a line chart we got a mountain chart

Okay, this is not perfect but believe me, we are getting there! Let's apply some aesthetic fixes on the chart and observe how it shapes up. Append the following to *styles.css*:

```
/* LINE CHART */
path {
fill: none;
stroke: #ed3700;
}
```

We need to set the fill to none to have the shapes reappear as lines. Refresh the graph:



The lines emerge

What separates a line chart from a bunch of lines stuck together on a graph? The ability to differentiate between the series. At the moment we only have the former.

For starters, we need to make a distinction between the lines in the code. Let's add an id to each line class – add the following to the LINES section in the Preparation part:

```
let id = 0;
const ids = function () {
   return "line-"+id++;
}
```

This little piece of code creates a counter that we can leverage to automatically assign a line id to every added line. Let's reference the counter in the class property of the paths. Adjust the code in the LINES section to add the class property:

```
const lines = svg.selectAll("lines")
    .data(slices)
    .enter()
    .append("g");

    lines.append("path")
    .attr("class", ids)
    .attr("d", function(d) { return line(d.values); });
```

And magically, each path gets its own class!

```
▼<g>
    <path class="line-0" d=
    "M0,328L73.84615384615385,310L147.6923076923077,276L221.53846153846155,3
    10L295.3846153846154,241L369.2307692307692,224L443.0769230769231,172L516
    .9230769230769,121L590.7692307692308,190L664.6153846153846,259L738.46153
    84615385,259L812.3076923076923,224L886.1538461538462,241L960,259">
    </path>
  </g>
▼<g>
    <path class="line-1" d=
    "M0,155L73.84615384615385,121L147.6923076923077,172L221.53846153846155,2
    07L295.3846153846154,224L369.2307692307692,172L443.0769230769231,138L516
    .9230769230769,69L590.7692307692308,86L664.6153846153846,155L738.4615384
    615385,190L812.3076923076923,190L886.1538461538462,155L960,138"></path>
  </g>
▼<g>
    <path class="line-2" d=
    "M0,224L73.84615384615385,190L147.6923076923077,138L221.53846153846155,1
    21L295.3846153846154,155L369.2307692307692,190L443.0769230769231,190L516
    .9230769230769,241L590.7692307692308,293L664.6153846153846,224L738.46153
    84615385,190L812.3076923076923,138L886.1538461538462,121L960,172">
    </path>
  </g>
```

The lines are given their own identity

What's left for us to do is to reference these classes in the css and give each line its own unique character. Change the Line Chart section of the css to say:

```
/* LINE CHART */
path.line-0 {
```

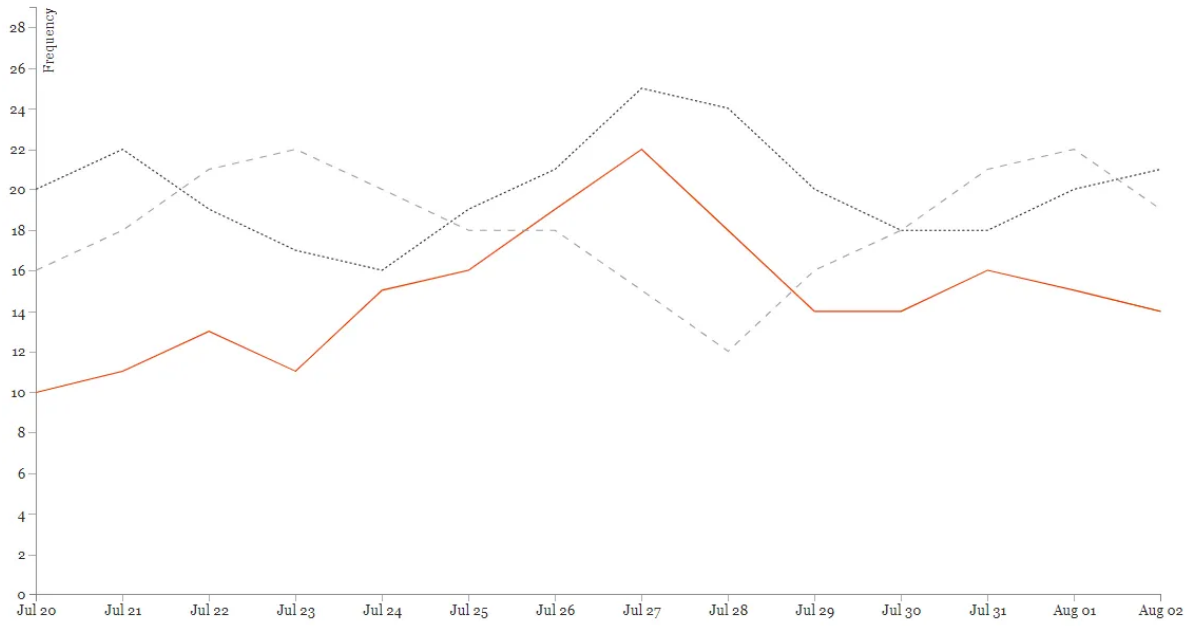
```
fill: none;
stroke: #ed3700;
}

path.line-1 {
fill: none;
stroke: #2b2929;
stroke-dasharray: 2;
}

path.line-2 {
fill: none;
stroke: #9c9c9c;
stroke-dasharray: 6;
}
```

Note how I'm not only amending the colour, but also changing the stroke of each line. Remember that about 10% of all people have some degree of colour blindness and in all fairness, differentiating between colours can be tricky for any of us. Colours will just blend together if there are too many data series and their hue will show differently on every monitor.

After the changes have been applied, the lines be clearly separated on the graph – like shown below:



The lines are visually different from each other

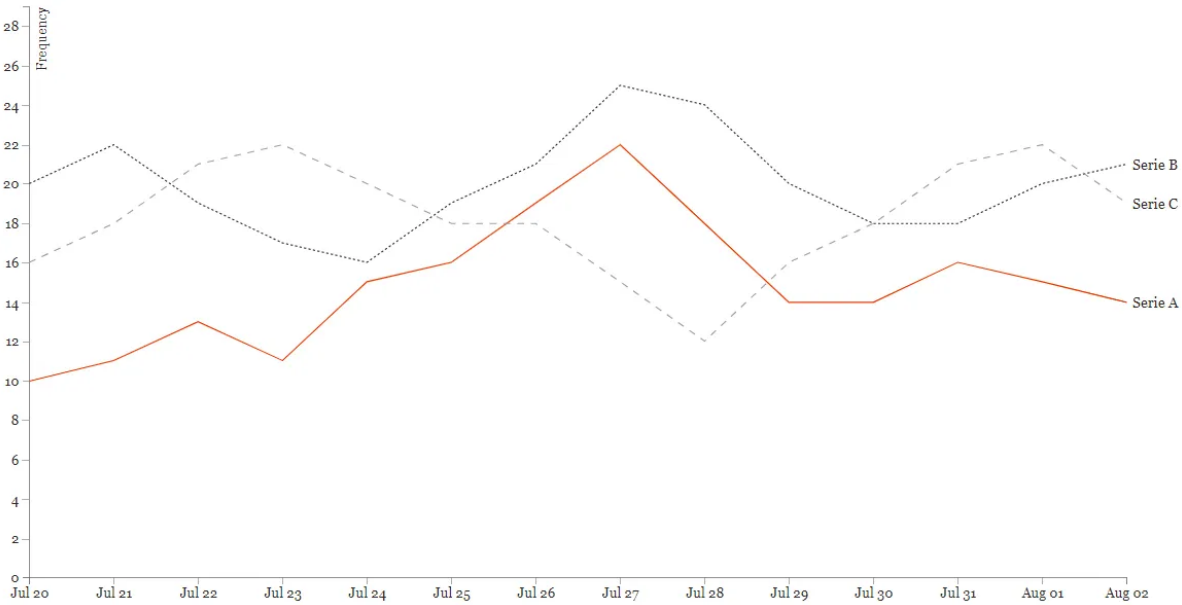
Now the series are differentiated but it's still impossible to tell which one is which unless you have memorised the underlying data and have a pretty sick visual imagination in which case I wonder why you needed a graph in the first place. To aid the majority of us in the series recognition I propose that we append the series name to the right side of the graph. Add the following to the drawing section of LINES:

```
lines.append("text")
   .attr("class","serie_label")
   .datum(function(d) {
      return {
         id: d.id,
         value: d.values[d.values.length - 1]}; })
   .attr("transform", function(d) {
         return "translate(" + (xScale(d.value.date) + 10)
         + "," + (yScale(d.value.measurement) + 5 ) + ")";})
   .attr("x", 5)
   .text(function(d) { return ("Serie ") + d.id; });
```

The snippet localises the end of each line and appends to it a text element. The text will be printed as Serie A, Serie B, or Serie C, depending on the line. Add the following to the css document to adjust the series labels:

```
.serie_label {
fill: #2b2929;
font-family: Georgia;
font-size: 80%;
}
```

The labels are appended! Good times.



Each series got its own label

We can all agree this is one handsome line chart! I've pasted the full code below. Make sure to check out the second part of the tutorial that presents two scenarios of adding interactivity to the chart.

line_chart.html:

```
<!DOCTYPE html>
<head>
<meta charset="utf-8">
<title>Multi Line Chart</title>
<script type="text/javascript" src="https://d3js.org/d3.v5.min.js"></script>
<link rel="stylesheet" type="text/css" href="styles.css">
<style></style>
</head>
<body>
<div id="container" class="svg-container"></div>
<script>
//------------------------1. PREPARATION-----------------------//
//-----------------------------SVG-----------------------------//
const width = 960;
const height = 500;
const margin = 5;
const padding = 5;
const adj = 30;
// we are appending SVG first
const svg = d3.select("div#container").append("svg")
   .attr("preserveAspectRatio", "xMinYMin meet")
   .attr("viewBox", "-"
        + adj + " -"
        + adj + " "
        + (width + adj *3) + " "
        + (height + adj*3))
   .style("padding", padding)
   .style("margin", margin)
   .classed("svg-content", true);

//----------------------------DATA----------------------------//
const timeConv = d3.timeParse("%d-%b-%Y");
const dataset = d3.csv("data.csv");
dataset.then(function(data) {
```

```
    var slices = data.columns.slice(1).map(function(id) {
      return {
        id: id,
        values: data.map(function(d){
          return {
            date: timeConv(d.date),
            measurement: +d[id]
          };
        })
      };
    });

  //----------------------------SCALES---------------------------//
  const xScale = d3.scaleTime().range([0,width]);
  const yScale = d3.scaleLinear().rangeRound([height, 0]);
  xScale.domain(d3.extent(data, function(d){
    return timeConv(d.date)}));
  yScale.domain([(0), d3.max(slices, function(c) {
    return d3.max(c.values, function(d) {
      return d.measurement + 4; });
      })
    ]);

  //----------------------------AXES----------------------------//
  const yaxis = d3.axisLeft()
    .ticks((slices[0].values).length)
    .scale(yScale);

  const xaxis = d3.axisBottom()
    .ticks(d3.timeDay.every(1))
    .tickFormat(d3.timeFormat('%b %d'))
    .scale(xScale);

  //----------------------------LINES---------------------------//
  const line = d3.line()
```

```
    .x(function(d) { return xScale(d.date); })
    .y(function(d) { return yScale(d.measurement); });

let id = 0;
const ids = function () {
    return "line-"+id++;
}
//-------------------------2. DRAWING--------------------------//
//----------------------------AXES----------------------------//
svg.append("g")
    .attr("class", "axis")
    .attr("transform", "translate(0," + height + ")")
    .call(xaxis);

svg.append("g")
    .attr("class", "axis")
    .call(yaxis)
    .append("text")
    .attr("transform", "rotate(-90)")
    .attr("dy", ".75em")
    .attr("y", 6)
    .style("text-anchor", "end")
    .text("Frequency");

//---------------------------LINES----------------------------//
const lines = svg.selectAll("lines")
    .data(slices)
    .enter()
    .append("g");

    lines.append("path")
    .attr("class", ids)
    .attr("d", function(d) { return line(d.values); });

    lines.append("text")
```

```
    .attr("class","serie_label")
    .datum(function(d) {
        return {
            id: d.id,
            value: d.values[d.values.length - 1]}; })
    .attr("transform", function(d) {
            return "translate(" + (xScale(d.value.date) + 10)
            + "," + (yScale(d.value.measurement) + 5 ) + ")"; })
    .attr("x", 5)
    .text(function(d) { return ("Serie ") + d.id; });

});
</script>
</body>
```

styles.css:

```
/* AXES */
/* ticks */
.axis line{
stroke: #706f6f;
stroke-width: 0.5;
shape-rendering: crispEdges;
}

/* axis contour */
.axis path {
stroke: #706f6f;
stroke-width: 0.7;
shape-rendering: crispEdges;
}
```

```
/* axis text */
.axis text {
fill: #2b2929;
font-family: Georgia;
font-size: 120%;
}

/* LINE CHART */
path.line-0 {
    fill: none;
    stroke: #ed3700;
}

path.line-1 {
    fill: none;
    stroke: #2b2929;
    stroke-dasharray: 2;
}

path.line-2 {
    fill: none;
    stroke: #9c9c9c;
    stroke-dasharray: 6;
}

.serie_label {
  fill: #2b2929;
  font-family: Georgia;
  font-size: 80%;
}
```

data.csv:

```
date,A,B,C
20-Jul-2019,10,20,16
21-Jul-2019,11,22,18
22-Jul-2019,13,19,21
23-Jul-2019,11,17,22
24-Jul-2019,15,16,20
25-Jul-2019,16,19,18
26-Jul-2019,19,21,18
27-Jul-2019,22,25,15
28-Jul-2019,18,24,12
29-Jul-2019,14,20,16
30-Jul-2019,14,18,18
31-Jul-2019,16,18,21
01-Aug-2019,15,20,22
02-Aug-2019,14,21,19
```

Share this:

Posted in D3, Data Visualisation

Previous Post

TUTORIAL: Advanced Bar Chart in D3.js v.5

Next Post

TUTORIAL: Making an Interactive Line Chart in D3.js v.5

Leave a Reply

Enter your comment here...

Search ...

Subscribe via Email

Enter your email address to subscribe to this blog and receive notifications of new posts by email.

Email Address

Subscribe