Column charts and their horizontal bar chart cousins are staples of every data visualization library ever created, but wouldn't you rather know how to create your own? You can learn how to build a chart like the one below in less time than it would take to find, download, and configure some pre-built ball of spaghetti code that may or may not actually meet your needs.

Data visualization in general and D3 in particular can be intimidating if you haven't worked with them before, but they don't have to be! In this short post you'll get annotated code samples, links to relevant documentation, and access to a working, editable example.

Let's get started!

Data to visualize

Before you can create a data visualization you need data, right? Usually you'll be loading JSON from a server but to minimize the moving pieces here we'll just use a plain old JavaScript Array of objects. We have five students and they each have properties for three different subjects.

```
const data = [
  { name: 'Alice', score: 37 },
  { name: 'Billy', score: 72 },
  { name: 'Cindy', score: 86 },
  { name: 'David', score: 44 },
  { name: 'Emily', score: 59 }
];
```

Define your dimensions

Next you want to define the size of your chart, including margins that will reserve space for your axes. The `width` and `height` properties are calculated by subtracting the margin values so you can use them later without having to think about margins or axes at all. `width` and `height` represent the actual usable space for your chart.

```
const margin = { top: 10, right: 10, bottom: 30, left: 30 };
const width = 800 - margin.left - margin.right;
const height = 600 - margin.top - margin.bottom;
```

Setting the stage

With the basics out of the way you can create the root of your chart, an SVG element, using the properties you defined above. When creating the `svg` tag you will add the margins back in, since it will be the parent element of the axes too. An SVG group, or `g` element, is then added and moved to the position defined by your `left` and `top` margins. The chart contents will be added to this group, ensuring they don't obscure or interfere with the axes.

```
const 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})`);
```

Tipping the scales

Now the stage is set for actually drawing our chart, but we have one thing left to do before we can start putting shapes on the screen. We need to define some scales, which is how D3 transforms input data like test scores into output values like pixel sizes.

The X scale will be a band scale, which is a special type of scale specifically for bar and column charts. Once the scale is created by calling `d3.scaleBand()`, the domain method is used to specify the set of input values. In this case, the values we want to display along the X axis are the student names, which we can get using a simple Array.map function. Next is the range method, where you specify the *range* of output values that the values in the input domain should be mapped to. Passing an array of `0` and our `width` property will tell D3 we want the chart to fill the available space. Lastly, the padding method will provide a bit of space between the columns on the chart. This is a normalized value, so `0` would be no space between columns, `0.5` would make the columns and gaps the same size, and `1` would be nothing but padding.

```
const xScale = d3.scaleBand()
  .domain(data.map(d => d.name))
  .range([0, width])
  .padding(0.2);
```

That takes care of data conversion for the horizontal dimension, but you probably want to label it as well. Since you want the X axis at the bottom of your chart, create a new `g` element and move it to the bottom using the `height` property defined previously. You'll then pass the `xScale` you created to `d3.axisBottom` since you want the labels below the axis line and tick marks.

```
svg
  .append('g')
  .attr('transform', `translate(0, ${height})`)
  .call(d3.axisBottom(xScale));
```

The Y scale is a linear scale, which just directly translates input values to output values. In this case the input domain is `[0, 100]` since `0` is the minimum score for a test and `100` is the maximum score. Similar to the X scale, the output range for the Y scale uses the `height` defined previously.

```
const yScale = d3.scaleLinear()
  .domain([0, 100])
  .range([height, 0]);
```

You may have noticed that this time the range array puts `0` last. Don't worry too much about the details, but this is essentially creating a "flipped" scale because in SVG, higher Y values are rendered further *down* on the screen. Since that runs counter to how we think about coordinates, namely that a higher score should result in a taller column, we invert the scale that controls column height.

Finally, we'll use `d3.axisLeft` to attach the axis this time, since we want the labels to the left of the axis.

```
svg
  .append('g')
  .call(d3.axisLeft(yScale));
```

Ready, set, draw

Now you're ready to draw the columns for your chart. Finally!

```
svg.selectAll('rect')
  .data(data)
  .enter()
  .append('rect')
    .attr('x', d => xScale(d.name))
    .attr('y', d => yScale(d.score))
    .attr('width', d => xScale.bandwidth())
    .attr('height', d => height - yScale(d.score));
```

`svg.selectAll('rect')` tells D3 to select all the `rect` (rectangle) elements that are children of the `svg` element. No, you didn't miss anything, we're trying to select elements that don't exist. Yet. The next line is referred to as a data join. We're telling D3 to merge the (empty) selection of `rect` elements with our `data` array of objects.

The data join creates a new selection, called the update selection. It's called the update selection because it's what we would use to update any existing `rect` elements, had there been any. Since we're starting with a blank slate, our update selection is empty and can be ignored.

We can access the enter selection by calling `enter()` on the update selection. The enter selection represents data items (objects from our array) without a corresponding DOM element. For those data items we'll create a new `rect` and append it to the DOM before setting all the attributes.

The call to `append(rect)`, unsurprisingly, adds the `rect` element to the `svg` element, but it also does something else important. It changes the context for any of the chained method calls that follow it. The subsequent calls to `attr()` will be called in the context of that new `rect` element, and the function passed as the second parameter is responsible for returning the value the attribute will be set to. The function will also receive the corresponding data item as its first argument, allowing you to set attribute values based on the underlying data. Literally, using data to drive the document, which is where the name D3 comes from.

The `x` and `y` attributes for positioning the columns are set by passing the appropriate data property to the scales we previously defined. The column `width` is set using the bandwidth method of our band scale, and the `height` is calculated using the height of the chart and our trusty `yScale`.

What's Next?

That's it! You built a column chart! I'd say a celebration is in order. Dance a jig, drink a tasty beverage, or simply sit back and marvel at what you've accomplished.

Oh, and if you feel like tinkering with a real live version, click here and have at it!

This one should track.

Want to learn more?

I've assembled this post and a few more like it into a guide called *D3 Quickstart: Your Guide to Awesome Visualizations*. It will show you step by step how to build column, bubble, line, and area charts. It also provides more information on some of D3's core concepts like selections and scales.

Drop your email address into the field below and it'll be sent to your inbox quicker than you can say Data-Driven Documents!

SEND ME THE GUIDE button