title: Building a bar chart component for dynamic data visualisation slug: building-dynamic-visualisation comonents description: Examines how to create a bar chart component for presenting data from a dynamic source. author: Jon Rimmmer tags: Angular, Visualisation, Component banner: ./images/banner.jpg bannerCredit: Photo by WHO!?

Given a requirement to visualise data, developers will often use a charting library. This isn't wrong, but it does involve tradeoffs: Each added dependency increases our app size, and may not be a perfect match for our requirements, requiring compromises or workarounds.

An alternative is build our own bespoke visualisation components. Modern HTML, SVG and CSS, combined with the power of a JavaScript framework like Angular, provides everything we need to do this, much more easily than used to be the case. In this article we'll look at doing just this, by creating our own bar chart component.

Basic Layout

When writing components, especially generic or reusable ones, we should think carefully about how we divide responsibilities between the component and its ***consumer***, which is typically the parent component in the tree. For example, for our bar chart, the component's repsponsibility is to display the data, and the parent's responsibily is to supply it.

In the case of CSS, it's easy to get this wrong, and hardcode decisions about size, color and position into the component that should be left up to the consumer. We want our component to be as responsive as possible, which means it must respect the layout mode of its parent, and render elastically into what space is given to it.

Stage 1: Basic Layout

When creating a

Stage 2: Connecting a data source

Stage 3:

Layout:

- Wherever possible, leave decisions about the size of the host element up to the consumer.
- Expose control of hardcoded values via customer properties.
- In a parent, don't fix the size of a child. Use a container such as flexbox or grid instead.

Dynamic data:

- Very easy to make a chart that looks good with carefully design sample data, or but soon breaks with "real" data.
- How many columns?
- How many bars within a column?
- Real data is messy:
    - Long names, without whitespace.
    - Weird ranges, it's never 0 to 100, it's more likely 99.5 to 100.0001.
    - Or it's huge, e.g. 4.9m to 8.56m

    - Or tiny: 0.0000001 to 0.0000002.
    - Just let the user configure it all? But sometimes that's not an option. When writing a tool for dealing with arbitrary datasets, we have to have default strategies.
        - Look great for some data.
        - Look OK for for almost all data.
        - Be at least functional for all data.

Angular support for CSS properties: https://github.com/angular/angular/issues/9343

Unfortunately, here we run into a limitation of Angular: it has very poor support for binding to modern CSS features like grid and custom properties.

In this article we'll build a bar chart component using Angular. You might expect us to use SVG for this, but SVG has the problem that, infinitely scalable, it does not have support for the kind of responsive layout we'd like our chart to exhibit. For example, given an SVG chart with an x-axis area and a column area...

...we have no way to say "keep the the x-axis area the same size, while scaling the chart area to take advantage of the available space".

If we were to render our chart to SVG, we would need to add a window resize listener and rerender things every time it fired. And this would only solve the problem of a whole-window resize. Other changes to the size of the host would not trigger a rerender without additional hacks.

Fortunately, there's an alternative, which is to leverage the excellent support of HTML + CSS for dynamic, responsive layout. By doing so, we can create a chart that sizes sensibly and fluidly to any dimensions, without the need for rolling our own layout engine. It might sound a little unorthodox, but stick with me.

Chart Structure

Our bar chart will consist of four main elements:

- The legend
- The x-axis
- The y-axis
- The data columns

We'll tackle each in turn, but the first thing we need to do is setup the basic layout of our component. As previously stated, we'll be using CSS grid, so we'll begin by adding some grid styles to `bar-chart.component.css`:

```
:host {
  display: grid;
  grid-template-rows: auto 1fr auto;
  // We'll make this dynamic later:
  grid-template-columns: auto 1fr;
}
```

Legend

The legend will be a simple list of the series in the chart.

Legend Items

Each legend item will display the series label, and the series color as a dot.

Since the dot color is part of the data, we need to bind it. What we'd ***like*** to do here is define out legend item as follows:

```
.legend-item::before {
  background-color: var(--itemColor);
  content: '';
  border-radius: 50%;
  // etc.
}
```

Here, we're using a CSS custom property to control the actual color of the legend item. Then, all we need to do supply that property in our template:

```
<li class="legend-item" [style.--itemColor]="s.color">{{ s.name }}</li>
```

Unfortunately, this does not work. Angular does not support binding CSS custom properties in this way. This has been an open issue for many years and doesn't seem likely to be fixed soon. It's a real shame, as binding properties in this way would let us have a much cleaner and more abstracted relationship between our CSS and our component.

While there are ways to get around this limitation and bind custom propreties, in this case, the easiest solution is to use a more explicit markup structure:

```
<li class="legend-item">
  <i [style-background-color]="s.color"></i>
  {{ s.name }}
</li>
```

Columns

Now we come to the real heart of our grid: the columns. Each column actually consists of two parts: the bars themselves, which visualise the magnitude of the data, and the label on the x-axis.

Grid Columns

Because the number of columns is flexible, depending on the data, our `grid-template-columns` style must be dynamic (grid does have support for auto-layout and implicit columns, but, for the sake of our sanity, we'll stick to defining our structure explicitly via bindings).

Our requirements aren't actually that complicated. We have a single x-axis column, followed by a number data columns:

[x-axis][data 1] [data 2] ... [data n]

In CSS grid syntax, that can be expressed as :

```
grid-template-columns: auto repeat(n, 1fr [col-start]);
```

The `[col-start]` is a grid-line name, and will let us easily target that column, without worrying about any other template columns around it.

So, we'll need a style host-binding for `grid-template-columns`:

```
@HostBinding('style.grid-template-columns')
public templateColumns = 'auto 1fr'; // Default if no data.

@Input()
public set data(value: BarChartData) {
  // ...
  this.templateColumns = `auto repeat(${ value.columns.length } 1fr [col-
start])`;
}
```

Unfortunately, once again, not so fast. If you try to run this in the browser, it won't work, and you'll get an error in the console warning about style sanitization. This is Angular's protection against cross-site-scripting being overzealous, not understanding that the `repeat(...)` function in our bound style is safe. This is another known problem with Angular's CSS binding capabilities.

To get around it, we need to tell Angular that this value is safe, which we can do using the `DomSanitzer` service. After adding this to our constructor, we can use it to mark the value as safe:

```
@HostBinding('style.grid-template-columns')
public templateColumns: string | SafeStyle = 'auto 1fr';

@Input()
public set data(value: BarChartData) {
  // ...
  this.templateColumns = this.sanitizer.bypassSecurityTrustStyle(
    `auto repeat(${ value.columns.length } 1fr [col-start])`
  );
}
```

There are alternatives to using the sanitizer, such as injecting the `ElementRef` and manipulating the native element's style directly, but the end result is the same.

Our component's host element will now be configured with the correct columns, and we can start putting stuff inside them.

Column Templates

As you might expect, we'll use the `ngFor` directive to loop through our column property and output some HTML for each one. Let's look at the finished markup, then break it down piece by piece:

```
<ng-container *ngFor="let column of columns; index as i">
  <div class="column-data"></div>
  <div class="colunm-label">
    {{ column.label }}
  </div>
</ng-container>
```

The first thing to observe is that we're looping on an `ng-container`, rather than an element. When using CSS grid, we need to keep our markup as flat as possible, because items can only be placed within a grid belonging to their parent element. This can feel a little strange at first, because we're so used to using nesting to group different types of elements. We might expect to have a `<div class="column">` with our `.column-data` and `.column-label` elements inside it. But doing this would prevent us placing each into separate grid cells.

> Note: In future, an enhancement to CSS Grid called sub-grid will allow more standard, nested mark-up structures, but for now it's just something we have to accept.