Revisiting grids

I was looking at Brad Frost's site to look at how he did the little circles at the top and bottom of the page. Hint: he does with a ton of SVG circles and some very interesting use of JavaScript to create the effect.

When I saw the CSS code I came accross multiple definitions for grids and that got me thinking, again, about automation and how to use CSS properties to make the code easier to work with.

CSS properties and Houdini custom properties

One of the problems about regular CSS variables (also known as CSS custom properties) is that athey are all represented as strings. This limits what we can do with them on JavaScript and how we use them in CSS.

CSS Properties and Values API Level 1 provides enhanced custom properties, defined in JavaScript, that overcome the deficiencies of the initial implementation of custom properties.

For more information about Houdini CSS Properties and Values see my other posts on the topic: CSS Houdini Properties & Values and CSS Houdini: Present and Future of CSS

We define three custom properties in JavaScript. We use feature detection to make sure that the browser supports Houdini Custom Properties before we add them.

For each property we register, we define four properties:

- **name** defines the name of the property that we'll use throughout the application
- **syntax** refers to the syntax for the element as defined in the supported syntax name section of the specification
- **inherits** tells the browser is the property should be inherited by the element's children. The default is false
- **initial value** gives the default value for the property

I place the definitions inside a feature query to see if the browser supports the `registerProperty` in the CSS object then we register the properties with the syntax below

```
if ('registerProperty' in CSS) {
  CSS.registerProperty({
    name: '--container-width',
 '--container-width'>',
    inherits: false,
    initialValue: 44,
  });

  CSS.registerProperty({
    name: '--grid-cell-size',
    syntax: '<integer>',
    inherits: false,
    initialValue: 200,
  });

  CSS.registe'--grid-cell-size'me: '--grid-gap-size',
    syntax: '<number>',
    inherits: false,
    initialValue: 1.5,
  });
} else {
  console.log('Houdini custom properties not supported');
  '--grid-gap-size'// Add non-Houdini custom features here
}
```

The layout container

```
/* Layout container */
.container {
  margin: 0 auto;
  padding: 0 1rem;
  max-width: calc(--var(--container-width) * 1rem)
```

```
  position: relative;
  z-index: 1;
}

/* Wide layout container variation */
.container--wide {
  --container-wdith: 79.2;
  max-width: calc(--var(--container-width) * 1rem);
}

/* Narrow layout container variation */
.container--narrow {
  --container-wdith: 37.4;
  padding: 0 1rem;
  margin: 0 auto 1rem;
  max-width: calc(--var(--container-width) * 1rem);
}
```

Defining the grids

Rather than defining the whole size of the grid in its definition, we only need to define those values that are common to all different types of grid we want to have ready as default.

When researching the structure of the default grids, I came across Auto-Sizing Columns in CSS Grid: `auto-fill` vs `auto-fit` by Sara Soueidan.

In this case I don't want to add new column cells to fit the available space so `auto-fit` is not the right answer. So we use `auto-fill` to keep the number of columns constat while make them fill the available space.

Here we use another variable to change the value of the minimum size for the cells. If we don't redefine the value of a variable then it uses the default value defined when we registered the variable.

```
/* Grid */
.grid {
```

```
  display: grid;
  grid-gap: var(--grid-gap-size);
}

.grid--small {
  grid-template-columns: repeat(auto-fill, minmax(var(--grid-cell-size), 1
  grid-template-rows: auto;
}

.grid--med {
  --grid-cell-size: 320;
  grid-template-columns: repeat(auto-fill, minmax(var(--grid-cell-size), 1
  grid-template-rows: auto;
}

.grid--loose {
  --grid-cell-size: 400;
  grid-template-columns: repeat(auto-fill, minmax(var(--grid-cell-size), 1
  grid-template-rows: auto;
}

.grid--gap-large {
  --grid-gap-size: 2.5;
  grid-gap: calc(var(--grid-gap-size * 1rem));
}
```

Using grids areas

One of the more esoteric aspects of the grid is that you can use grid template areas to associate elements with it and then define those areas using an ASCII-like layout.

The first step is to associate elements with a `grid-area` element.

```
.header {
    grid-area: hd;
}
```

```
.footer {
    grid-area: ft;
}
.content {
    grid-area: main;
}
.sidebar {
    grid-area: sd;
}
```

Next we define the grid. Working with template areas requires one additional step.

For each cell that we want to use we need to define what area we want to place it in by naming the area. In the example below we associate each cell of our 9-column grid with one of the areas we defined earlier.

If we want to leave a cell blank then use a period (`.`) in the cell definition.

```
.wrapper {
  display: grid;
  grid-template-columns: repeat(9, 1fr);
  grid-auto-rows: minmax(100px, auto);
  grid-template-areas:
    "hd hd hd hd   hd   hd   hd   hd   hd"
    "sd sd sd main main main main main main"
    "sd sd sd main main main main main main"
    ".  .  .  ft   ft   ft   ft   ft   ft";
}
```

Once we've created the areas and assigned cells to them we can use them in regular HTML like the code below. this will automatically create the layout we specified with the `grid-template-areas` descriptor.

```
<div class="wrapper">
    <div class="header">Header</div>
    <div class="sidebar">Sidebar</div>
    <div class="content">Content</div>
```

```
    <div class="footer">Footer</div>
</div>
```