Revisiting Aart Directed Layouts

I've always loved the work that Andrew Clarke has done over the years with art direction on the web. I came accross another of his ideas in `Inspired Design Decisions For The Web #3` that was part of my Smashing Magazines membership.

In it he talks about how we can make the same basic layout more interesting for our readers without having too many additional libraries.

It doesn't have to be boring

For ages we've blamed CSS frameworks for the sameness of our designs. We think that Bootstrap or Foundation are what make our designs bland and repetitive.

I haven't used Foundation and Bootstrap in a while and Foundation (the framework I've used the most) has already implemented CSS Grid but Boostrap is still using a Flexbox layout for its grids.

If we're willing to trade backwards compatibility and the need for polyfills for ease of development we can use native CSS Grids, explained by Rachel Andrew and specified by the CSS Working Group

In it's most basic form we can create a grid with this bit of CSS:

```
.page-container {
  display: grid;
  grid-template-columns: repeat(12, 1fr);
  grid-gap: 10px;
}
```

With this, all children of `page-container` are now items on the grid we just defined.

To add a little more flexibility I use custom properties to define the number of columns and the value of the gap.

With those changes the baseline code looks like this:

```
:root {
  --grid-columns: 12;
  --grid-gap: 10;
}

.page-container {
  display: grid;
  grid-template-columns: repeat(var(--grid-columns), 1fr);
  grid-gap: calc(var(--grid-gap)* 1px);
}
```

Just by changing the value of `--grid-columns` in the `:root` pseudo element we can change the number of columns (that will still take equal portioons of the space on screen) and the gap between those columns.

With this flexibility we can create custom just by redefining the variables that we want to change. For example a 6-column grid in the same page could look like this:

```
.six-col-grid {
  --grid-columns: 6;
  --grid-gap: 10;

  display: grid;
  grid-template-columns: repeat(var(--grid-columns), 1fr);
  grid-gap: calc(var(--grid-gap)* 1px);
}
```

We redefine the columns inside the class where we want to change the values and the codee remains the same.

Fit content in the grid: Basic Placement

Now that we've created the grid we can place contents in it.

CSS Grid has an implicit algorythm for placing items when there are no explicit instructions for placement.

If you create a grid all child items will lay themselves out one into each grid cell. The default flow is to arrange items by row. Grid will lay an item out into each cell of row 1. If you have created additional rows using `grid-template-rows` then grid will continue placing items in these rows. If the grid does not have enough rows in the explicit grid or if you did not create explicit rows at all it will creaate new implicit rows.

For example, reusing the grid from our last section.

```
:root {
  --grid-columns: 12;
  --grid-gap: 10;
}

.page-container {
  display: grid;
  grid-template-columns: repeat(var(--grid-columns), 1fr);
  grid-gap: calc(var(--grid-gap)* 1px);
}
```

The HTML code below will produce twelve items in row one and nine items in row two.

```
<div class="page-container">
  <div class="box 1">1</div>
  <div class="box 2">2</div>
  <div class="box 3">3</div>
  <div class="box 4">4</div>
  <div class="box 5">5</div>
  <div class="box 6">6</div>
  <div class="box 7">7</div>
  <div class="box 8">8</div>
  <div class="box 9">9</div>
  <div class="box 10">10</div>
  <div class="box 11">11</div>
```

```
  <div class="box 12">12</div>

  <div class="box a">a</div>
  <div class="box b">b</div>
  <div class="box c">c</div>
  <div class="box d">d</div>
  <div class="box e">e</div>
  <div class="box f">f</div>
  <div class="box g">g</div>
  <div class="box h">h</div>
  <div class="box i">i</div>
</div>
```

You can see a working version in this Codepen sample.

This is cool but it's too limiting to let the explicit algorythm do all the work.

We are able to place items in specific locations on the grid, taking advantage of the implicit behavior where needed, without restrictions.

For this example we'll use fewer items in our container. It looks like this.

```
<div class="page-container">
  <div class="box box1">1</div>
  <div class="box box2">2</div>
  <div class="box box3">3</div>
  <div class="box box4">4</div>
</div>
```

The grid definition and the variables we used to create it remain unchanged. We've added the following rules to actually place the boxes in the grid at the specified locations.

```
.box1 {
  grid-column: 4 / 6;
  grid-row: 2 / 3;
}
```

```
.box4 {
  grid-column: 1 / 2;
  grid-row: 1 / 3;
}

.box2 {
  grid-column: 5 / 9;
  grid-row: 7;
}

.box3 {
  grid-column: 10 / 12;
  grid-row: 1;
}
```

Notes

Wherethere are two numbers for grid-column or grid-row it indicates a span, the first number is the starting poisition and the second the latest.

While we can move the content around for visual displays this will not change the document order (there is another attribute for that). However, make sure that you can read the document in source order and that it makes sense for the reader.

Fit content in the grid: Named Lines

Another way to create the templates is to give each column a name. The code below creates 12 columns, each named `col-start` witha corresponding `col-end`. We're keeping 10px gaps between the columns.

```
:root {
  --grid-columns: 12;
  --grid-gap: 10;
}

.page-container {
```

```
  display: grid;
  grid-template-columns:
    repeat(var(--grid-columns), [col-start] 1fr [col-end]);
  grid-gap: calc(var(--grid-gap)* 1px);
}
```

This presents an interesting issue. If all columns are named the same how can we reference a specific instance?

We can take the name of the column and the number represeting the position.

```
.box1 {
  grid-column: col-start 4 / col-start 6;
  grid-row: 2 / 3;
}

.box4 {
  grid-column: col-start / col-start 2;
  grid-row: 1 / 3;
}

.box2 {
  grid-column: col-start 5 / col-start 9;
  grid-row: 7;
}

.box3 {
  grid-column: col-start 10 / col-start 12;
  grid-row: 1;
}
```

This Pen shows how these columns work and how we've produced the same display using named lines.

Making the job easier: Using grid template areas

Therre is one last way to create layouts using grid: Template Areas.

A grid area is the logical space used to lay out one or more grid items. A grid area consists of one or more adjacent grid cells. It is bound by four grid lines, one on each side of the grid area, and participates in the sizing of the grid tracks it intersects. A grid area can be named explicitly using the `grid-template-areas` property or referenced implicitly by its bounding grid lines.

```
.grid-container  {
  display: grid;
  grid-template-columns: repeat(6, 1fr);
  grid-template-rows: 100px 1fr 1fr 100px;
  grid-template-areas: "a a a a a a"
 "a a a a a a"         "b b b c c c"
                       "b b b c c c"
                       "b b b c c c";
}

"b b b c c c".item1 { grid-area: a; }
.item2 { grid-area: b; }
.item3 { grid-area: c; }
```

The other thing worth noting this is how to leave areas of the template blank. In our definition of the template area use a dot (`.`) to designate areas where we don't want content.

In our modified example, we've narrowed out content area to ignore the first column in area b and the last item in area c.

```
.page-container  {
  display: grid;
  grid-gap: 10px;
  grid-template-columns: repeat(6, 1fr);
```

```
  grid-template-rows: 100px 1fr 1fr 100px;
  grid-template-areas: "a a a a a a"
 "a a a a a a"         ". b b c c ."
                       ". b b c c ."
                       "d d d d d d";
}

". b b c c .".box1 { grid-area: a; }
.box2 { grid-area: b; }
.box3 { grid-area: c; }
.box4 { grid-area: d; }
```

The resulting code looks like this pen

Media Queries And Making It Work On Mobile

We've worked on building the grids but these are not responsive by default. We can use media queries to make the layout responsive by doing one or both of the following:

- change the layout itself
- change the way that content is laid out on different form factors

In this example we have two media queries that will change the number of columns based on the device's width. If the screen is 768px (48em) or narrower, we want to work with three columns.

If the screen is wider than 48em then we want to work with 12 columns.

This is meant as an example, I would probably add more media queries to acommodate larger and smaller form factors.

```
/* These are global defaults */
:root {
  --grid-columns: 12;
  --grid-gap: 10;
```

```
}


@media screen and (max-width: 48em;) {
  .page-container {
    /* Override :root custom properties */
    --grid-columns: 3;
    --grid-gap: 10;
    display: grid;
    /*
      The overrides will use the new values
      in the declarations below
    */
    grid-template-columns: repeat(var(--grid-columns), 1fr);
    grid-gap: calc(var(--grid-gap)* 1px);
  }
}

@media screen and (min-width: 48em) {
  .page-container {
    display: grid;
    /*
      These declarations will use the values from :root
    */
    grid-template-columns: repeat(var(--grid-columns), 1fr);
    grid-gap: calc(var(--grid-gap)* 1px);
  }
}
```

The second part is to change the way we layout the content.

In this version, we move the sidebar to the bottom of the page in smaller screens and we reduce the number of columns to make sure that the content remains readable.

Here we setup a basic default for the `page-container` element.

```
.page-container {
```

```
  display: grid;
  grid-gap: 1em;
  grid-template-areas:
   "title"
   "sidebar"
   "content"
   "sidebar2"
   "footer"
}
```

We also set up two media queries. One for screens smaller than 48em (768px) and one larger.

The first query is for the smaller screens. Here we've moved the sidebars under the main text.

```
/* screen smaller than 48em */
@media screen and (max-width: 48em)  {
  .page-container {
    grid-template-columns: repeat(3, 1fr);
    grid-gap: 10px;
    grid-template-areas:
    "title title title"
    "content content content"
    "sidebar sidebar sidebar"
    "si"title title title"ebar2"
    "footer footer footer";
  }
}
```

The second media query sets up our standard default for screens larger than 48em. We put the sidebars to each side of the main content.

```
/* screen larger than 48em */
@media screen and (min-width: 48em)   {
  .page-container {
    grid-gap: 20px;
```

```
    grid-template-columns: 120px auto 120px;
    grid-template-areas:
    "title  title  title"
    "sidebar content sidebar2"
    "sidebar content sidebar2"
   "title  title  title"ter";
  }
}
```

There is a working Codepen that shows how it works.

Putting it all together

Following `Inspired Design Decisions` and Art Directing For The Web With CSS Grid Template Areas we can start looking at how to build more engaging and communicative layouts for our web content.

Because they are all part of the web platform, we can combine grids with other layout methods like multi-column or flexbox to creates powerful layout possibilities.

This is a very basic layout meant as a starting point to experiment with and continue evolving. I'll explain more about this in the conclusion.

I've broken the code into sections to make the explanation easier to follow.

The first block deals with utilities that we'll use throughout the rest of the stylesheet.

The first thing we do is import the font we'll use from Google Fonts. Normally I would use local fonts rather than importing them like this but I'm ok with the potential rendering delay because it's an experiment and because Google fonts now uses **font-display: swap** by default.

Next we make sure that all elements of the page use **box-sizing: border-box** to eliminate border and margins as sources of potential layout issues later.

The last thing in this block is to setup default for our font in the `html` element.

```
@import url('https://fonts.googleapis.com/css?family=Lato:400,400i,700,700

*,
*:before,
*:after {
  box-sizing: border-box;
}

html {
  font-size: 16px;
  font-family: Lato, sans-serif;
  font-weight: normal;
  font-style: normal;
}
```

The next block creates classes for displaying images 2 at a time in the same row. I know I can probably get away without these classes but I wanted something that would hold without resizing or changing the images' location.

```
.image-2-up {
  display: flex;
  flex-direction: row;
  align-content: stretch;
}

.child-image {
  width: 100%;
  height: 100%;
  margin: auto;
}
```

Next we create our base style for the `grid-container` element. These are the defaults that we'll override with media queries later.

Along with the defaults, this block also sets up the grid template areas that each type of element will use. This is pretty much set in stone as what will change is the grid template itself in the media queries.

```
.page-container {
  display: grid;
  grid-gap: 1em;
  grid-template-areas:
   "title"
   "con"title"  "content"
   "footer"
}

"content".title {
  grid-area: title;
}

.content {
  grid-area: content;
}

.image-2-up {
  grid-area: figures;
}

.footer {
  grid-area: footer;
}
```

The first media query is for window sizes larger than 48em. In this case we set up a 6-column grid and place the title and the content in different locations and with different column widths.

We also make the content a single row of that spans across the layout.

We place the title and content so they start in different locations to create more white space and make it easier to read.

```
@media (min-width: 48em)   {
  .page-container {
    grid-gap: 20px;
    grid-template-columns: repeat(6, 1fr);
```

```
    grid-template-areas:
    "title  title  title title  title  title"
    "title  title  title title  title  title"figures"
    "content content content content content content"
    "footer  footer  footer footer footer footer";
  }
}

"content content content content content content".title {
  grid-column: 2 / 4;
  font-size: 150%;
}


.content {
  grid-column: 2 / 6;
  columns: 2;
  column-gap: 2.5em;
}
```

The last query is for screen sizes smaller than 48em. In this case we make more radical changes. We switch from grid to block layouts to make sure we can view as much content as possible in smaller devices.

The images flex box is changed from row orientation to column so that the images will display in a vertical line rather than horizontal.

And we switch the content to a single column. For some reason, setting `columns` to 1 is not enough. We also have to make the column width 100% for it to work.

```
@media (max-width: 48em)  {
  .page-container {
    display: block;
    margin: 1em;
  }

  .image-2-up {
```

```
    display: flex;
    flex-direction: column;
  }

  .content {
    columns: 1;
    column-width: 100%;
  }
}
```

That's it. There is a working Codepen that shows what the code does and how it works.

Closing

The code shown here is a good starting point but it's not complete by any stretch of the imagination. Here are a few things that I'd like to further explore.

Does the `image-2-up` flexbox scale to more than 4 images? what about smaller images?

I took 2 images and stitched them together using this class. Is it flexible enough to handle more than 2 images, different sized images or an odd number of images?

How to revert multi column layout to a single column

Better typography

Because I was more concerned with layout for the purpose of this post, I didn't pay attention to typography. There are things that we could do to make the text read easier and that's something I'd like to explore further, particularly in the multi column layout.

- Art Directing For The Web With CSS Grid Template Areas
- Revolutionize Your Page: Real Art Direction on the Web0 Presentation Deck by Jen Simmons
- Real Art Direction on the Web - Conference talk by Jen Simmons
- [Grid By Example]