Notes - CSS - Common Grid Layouts

- Dr Nick Hayward

A general guide to common CSS Grid layouts.

- https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Grid_Layout/Realizing_common_layouts_using_CSS_Grid_Layout

Contents

Intro

Grid layout with CSS is useful for structure and organisation in a HTML page. Its usage may be considered akin to previous layout options with tables, thereby enabling a developer to add columns and rows to a page.

However, a grid layout enables more complex and interesting layout options, including overlap and layers.

Sample layouts

As we design web apps using an underlying CSS Grid structure, we may find common layout options and designs particularly useful.

For example,

- responsive layouts - common option for web app on desktop, tablet, and mobile phone
- auto placement for dynamic content and media
- ...

Responsive layout

Display a layout with a variety of patterns and structures, e.g.

- single column for a phone
- add a sidebar for a tablet of lower window resolution

- full width view with dual sidebars &c.

page structure

We can start with a sample page structure for a HTML page,

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>CSS Grid - Responsive Layout</title>
    <link rel="stylesheet" type="text/css" href="./assets/style.css">
  </head>
  <body>
    <div class="wrapper">
                        ...
                </div>
        </body>
</html>
```

Then, we'll add some HTML5 markup for a `header`, `navigation`, `footer`, and some `main` content.

```
<div class="wrapper">
        <header class="site-header">
                <h3>Spire & the Signpost</h3>
                <h5>Shine through the gloom, and point to the stars...
</h5>
        </header>
        <nav class="site-nav">
                <ul>
                        <li><a href="">Home</a></li>
                        <li><a href="">Charts</a></li>
                        <li><a href="">Data</a></li>
                        <li><a href="">Views</a></li>
                </ul>
        </nav>
        <!-- use aside for tangentially related content for parent
section... -->
        <aside class="content-side">
                <header>
                        <h5>sidebar...</h5>
                </header>
        </aside>
        <main class="content">
                <article class="content-article">
                        <header class="article-header">
                                <h5>Welcome</h5>
                        </header>
                        <p>...</p>
                </article>
```

```
        </main>
        <section class="site-links">
                <h6>social links...</h6>
        </section>
        <footer class="site-footer">
                <h6>footer...</h6>
        </footer>
</div>
```

css and structure

For the above page structure, we also need to define some template areas for our grid in the CSS.

e.g.

```
/* CONTENT */
.content {
        grid-area: content;
}
```

We can use such template area names, defined with the `grid-area` property, to define a layout for the overall page, or part of a page.

These template areas may then be used with the parent for the grid structure, `wrapper`, in this instance for the overall page.

e.g.

```
.wrapper {
        display: grid;
        grid-gap: 10px;
        grid-template-areas:
                "site-header"
                "site-nav"
                "content-side"
                "content"
                "site-links"
                "site-footer"
}
```

So, the `wrapper` class will display as a grid, with a gap between each area of the grid. It only has a single column in this example, which includes the required order for the grid areas.

define media query

This grid structure might be suitable for a mobile phone with a single column view. However, it will currently also display for all devices and resolutions.

We can add a ***media query*** for initial alternative displays, which may be triggered using the width of the available screen real estate.

e.g.

```
/* min 700 */
@media (min-width: 700px) {
        .wrapper {
                grid-template-columns: 1fr 3fr;
                grid-template-areas:
                "site-header site-header"
                "site-nav site-nav"
                "content-side content"
                "site-links site-footer"
        }
}
```

specific media query

We may add further media queries to handle various rendering requirements. For example,

```
@media (min-width: 700px) {
        .wrapper {
                grid-template-columns: 1fr 3fr;
                grid-template-rows: 120px 60px calc(98vh - 240) 60px;
                grid-template-areas:
                "site-header site-header"
                "site-nav site-nav"
                "content-side content"
                "site-links site-footer";
                height: 98vh;
        }
}
```

This media query adds a `height` property to push and fix the footer at the bottom of the page.

We specify the height of the current ***viewport*** using a relative unit, `vh`.

We also add `grid-template-rows` to define known heights for three of the four rows, and a variant height for the main content. However, the main content is only given a height corresponding to available space in the viewer window.

This is achieved using the `calc()` function.

relative lengths

As noted above for the `height` property, we may use relative lengths and calculations for CSS property values.

For example,

- `vw` - variable width relative to 1% of width of current viewport
- `vh` - variable height relative to 1% of height of current viewport
- `vmin` - relative to 1% of viewport's smaller dimension
- `vmax` - relative to 1% of viewport's larger dimension

sample updates for responsive layout

After testing a responsive layout, we might add the following updates to a working sample layout.

- create a parent banner area for a header, user login, site search, and site nav

```
.banner {
        grid-area: site-banner;
        display: grid;
        grid-template-columns: auto 300px;
        grid-template-rows: 120px 60px;
        grid-template-areas:
                "site-header banner-extras"
                "site-nav site-nav";
}
```

This helps us easily manage the layout and relative sizes of banner content, regardless of page width and height. So, `banner-extras` might be styled as follows,

```
.banner-extras {
        grid-area: banner-extras;
        display: grid;
        grid-template-areas:
                "site-user"
                "site-search";
        padding: 5%;
}
```

Use of a child grid helps us manage fixed places within the parent `banner` area.

- media query for `min-width` of 900px with checks

```
/* min 900 */
@media (min-width: 900px) {
        .wrapper {
                grid-template-areas:
                "site-banner site-banner"
                "content-side content"
                "site-links site-footer";
                height: 98vh;
                grid-template-columns: 250px 3fr;
```

```
                grid-template-rows: 180px auto 60px;
        }
}
```

Auto placement for dynamic content and media

We can also use CSS grid with Flexbox to create content layouts similar to placing cards in the UI.

For example, we might create a layout to dynamically render images for a photo album, or a series of products in a brochure.

We can start by defining a simple list with various list items, e.g.

```
<ul class="items">
        <li>One</li>
        <li>Two</li>
        <li>Three</li>
        <li>Four</li>
        <li>Five</li>
        <li>Six</li>
        <li>Seven</li>
        <li>Eight</li>
        <li>Nine</li>
</ul>
```

We might then render these list items in flexible columns within our grid layout. We can define a minimum size, and then ensure that they expand to equally fill available space. This will ensure our rendered layout includes equal width columns regardless of the available content.

e.g.

```
/* content items */
.items {
        display: grid;
        grid-gap: 5px;
        grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
        list-style: none;
}
```

and render individual items using flexbox

```
.items li {
        border: 1px solid #3b8eb4;
  display: flex;
  flex-direction: column;
}
```

Fun layout - a game board

We can also use a grid layout for internal uses, such as a game board.

For example, HTML for such a 3x3 board might be as follows

```
<main class="content">
        <ul class="items">
                <li>
                        <h5>One</h5>
                </li>
                <li>
                        <h5>Two</h5>
                </li>
                <li>
                        <h5>Three</h5>
                </li>
                <li>
                        <h5>Four</h5>
                </li>
                <li>
                        <h5>Five</h5>
                </li>
                <li>
                        <h5>Six</h5>
                </li>
                <li>
                        <h5>Seven</h5>
                </li>
                <li>
                        <h5>Eight</h5>
                </li>
                <li>
                        <h5>Nine</h5>
                </li>
        </ul>
</main>
```

In this example, we use each list item as a square on the 3x3 board.

We may create the grid for the `content` class as follows,

```
/* CONTENT */
.content {
        grid-area: content;
        display: grid;
        grid-template-areas:
                "items";
        grid-template-columns: 1fr;
```

```
        align-self: center;
        justify-self: center;
        align-items: center;
        padding: 50px;
        border: 1px solid #aaa;
}
```

We can embed this content area within other grids, and then add child items for the grid content itself. The `content` container itself will be aligned and justified to the centre of the parent.

Each child column will occupy the same proportion of available space, `grid-template-columns: 1fr`. Each item will also be aligned to the centre of the available space.

Properties such as padding and border are optional, and dictated by aesthetic requirements.

grid items for the board

Each square will be a child list item to the parent `ul`, which is styled as follows

```
.items {
        grid-area: items;
        display: grid;
        grid-gap: 10px;
        grid-template-columns: repeat(3, 150px);
  grid-template-rows: 150px 150px 150px;
}
```

We can then style each item, which creates the squares on the game board. e.g.

```
.items li {
        margin: 0;
        list-style-type: none;
        border: 1px solid #333;
        background-color: #333;
        color: #fff;
        padding: 10%;
}
```

This styling is for aesthetic purposes to render a list item as a square without the default list style. We may also define an alternating colour scheme for our squares, e.g.

```
.items li:nth-child(even) {
        border: 1px solid #ccc;
        background-color: #ccc;
        color: #333;
}
```

References

- MDN - CSS Grid
- MDN - CSS Grid Common Layouts
- MDN - Flexbox
- W3 - CSS Units