Building a stylesheet using cascade layers

Note:

Credit for a lot of information and ideas on this post goes to Miriam Zuzanne's A Complete Guide to CSS Cascade Layers publlished in CSS Tricks.

CSS layers present an interesting design paradigm. They allow developers to group styles based on criteria we define.

For the example, we'll define four layers.

```
@layer
  base
  layout
  utilities
  theme;
```

The order we declare our layers is important, it will dictate how they cascade.

In our example, the rules from the `theme` layer will have precedence over the rules in the `layout` layer.

Using the layers we defined earlier, the cascade precedence order (from more to least important) is:

1. Styles outside a layers
2. base
3. layout
4. utilities
5. theme

The !important property will change the order of properties in @layers as it does in regular stylesheets using `!important` the order of precedence will be reversed before we apply our regular layer ordering

1. `!important` theme
2. `!important` utilities
3. `!important` layout
4. `!important` base
5. base
6. layout
7. utilities
8. theme

`revert-layer`

The `rever-layer` keyword allow us to revert the property to the value held in the previous `@layer`.

In the example below, the `no-theme` class will revert the color to the previous layer, setting it green.

```
@layer default {
  a { color: green; }
}

@layer theme {
  a { color: purple; }

  .no-theme {
    color: revert-layer;
  }
}
```

Adding rules to layers

Since we declared the project layers at the top of the document and set their precedence order, we then add rules to individual layers by calling the `@layer` at-rule with the name of the layer we want to place the rules in and then add as many rules as we need to.

```
@layer utilities {
  .padding-lg {
    padding: .8rem;
  }
}
```

We can add multiple rules to each layer and we can repeat the process multiple times. Each time it will append the new rules to the layer.

Adding entire stylesheets to a layer

We can also import stylesheets directly to a @layer using the @import at rule with a `layer` attribute indicating what `@layer` the browser should attach the stylesheet to.

This is particularly useful when using third-party scripts or when working with a modular architecture.

In this example, we add a local copy of normalize.css to the `reset` @layer.

```
@import url('normalize.css') layer(reset);
```

Nesting layers

we can also nest layers:

```
@layer defaults {
  /* Ordering the sublayers */
  @layer reset, typography;

  @layer typography {
    /* Styles go here */
  }

  @layer reset {
    /* Styles go here */
```

```
  }
}
```

We can reference these nested layers using a dot notation like `defaults.reset` and `defaults.typography`.

The rules of layer-ordering apply at each level of nesting. Any styles that are not further nested are considered “un-layered” in that context, and have priority over further nested styles:

```
@layer defaults {
  :any-link { color: green; }

  /* layered defaults (lower priority) */
  @layer reset {
    a[href] { color: red; }
  }
}
```

Mixing with existing content

My biggest issue is what to do if we’re implementing layers with existing code.

All styles outside of layers are put in an implicit layer at the end of the document, making these styles the ones with the highest priority and will override all content in layers.

```
h1 {
  color: green;
}

@layer layer-1 { h1 { color: red; } }
@layer layer-2 { h1 { color: orange; } }
@layer layer-3 { h1 { color: yellow; } }
```

We can build get around this limitation by adding layers before the layers created

for the framework. The example below uses a lower layer to override `!important` styles from the framework, and a higher layer to override normal styles.

```
@layer
  bootstrap.important,
  bootstrap.bootstrap,
  bootstrap.local;

@import url('bootstrap.css') layer(bootstrap.bootstrap);

@layer bootstrap.local {
  /* most of our normal bootstrap overrides can live here */
}

@layer bootstrap.important {
  /*
    add !important styles in a lower layer
    to override any !important bootstrap
    styles
  */
}
```

Why would we want @layers

Layers have some interesting use cases. We've discussed some already so I'll cover one that is think an important one: building a CSS architecture.

Building a CSS architecture

I think I finally came to understand the way that `@layers` work and, in the process, I built my own structure based on my project's needs. Something like this from generic to specific:

1. default styles (these could be nested or not)
    1. reset or normalize scripts
    2. base styles for components if no design system is applies
    3. typography
2. External styles from libraries or design system

3. themes
4. overrides

In the end, how you structure your layers is up to you and what makes the most sense for your project.

Links and resources

- @layer — MDN
- Cascade Layers — Chrome Developers
- A Complete Guide to CSS Cascade Layers
- Hello, CSS Cascade Layers