Animations are great. I covered them back in our first AtoZ CSS series and now we'll look at the Sassy variety too!

Keyframes Recap

Let's start off with a quick recap of how CSS animations work.

Most CSS animations change a series of style properties over time by specifying a series of keyframes with a percentage syntax.

For example, we could animate the `left` position of an element from `0` to `100%` and back again using the following snippet:

```
@keyframes move {
    0%   { left:0; }
    100% { left:100%; }
}
.logo {
    position:absolute;
    top:0;
    left:0;

    width:100px;
    height:100px;
    border:1px solid;

    animation: move 5s alternate 4 ease;
}
```

First we have a block of keyframes which describe the key moments in time throughout the duration of the animation.

Then we apply these keyframes to an element (or multiple elements) by setting the shorthand `animation` property. In this case this is made up of the `animation-name`, `animation-duration`, `animation-direction`, `animation-iteration-count` and `animation-timing-function` properties.

With this type of animation, we specify the styles for key points in the animation and the browser works out all the in-between states automatically.

Animation Mixin

As we've just seen, animations are comprised of a set of keyframes and the animation properties.

Since the two go hand in hand, we can create a Sass mixin to clean up our code a bit. Let's work through a staged example to build out the mixin and then refactor it to make it as flexible as possible.

Let's continue with the simple example we've just been looking at.

```
@keyframes move {
    0%   { left:0; }
    100% { left:100%; }
}
.logo {
    animation: move 5s infinite alternate ease;
}
```

As we want to create the keyframes and use them at the same time, we could create a mixin that takes arguments for each of the various animation properties and then uses the `@content` directive so we can create the keyframes in the body of the mixin.

```
@mixin animation( $name, $duration, $delay, $iteration, $direction, $fill-mode, $timing ) {
    @keyframes #{$name} {
        @content;
    }
    animation-name: $name;
    animation-duration: $duration;
    animation-delay: $delay;
    animation-iteration-count: $iteration;
    animation-direction: $direction;
    animation-fill-mode: $fill-mode;
    animation-timing-function: $timing;
}
```

This provides an interface to create the block of named keyframes and then pass in all the animation properties at once.

We'd use this mixin as follows:

```
.logo {
    @include animation( move, 5s, 0, infinite, alternate, forwards, ease) {
        0% { left:0; }
        100% { left:100%; }
    }
}
```

This works but there are some limitations. Firstly, we have to specify each of the parameters in the mixin when calling it with `@include`. Or

specify a `null` value if we want to skip one.

Secondly, we have to remember the correct order of parameters when using the mixin. This could be fraught with human error and we can do better by making our mixin more flexible by making some parameters optional.

We can do this by specifying default values of `null` in the mixin definition. We'd now only have to specify a name and a duration to use the mixin which is a slight improvement.

```
@mixin animation( $name, $duration, $delay:null, $iteration:null, $direction:null, $fill-mode:null
}
@include animation( move, 5s ) {
    0% { left:0; }
    100% { left:100%; }
}
```

But this still doesn't solve the problem of having to remember the order of the parameters. Instead, we can construct our mixin to accept a variable number of arguments and use the `animation` shorthand property to apply them to our element.

We do this with a `...` syntax after the last parameter name.

```
@mixin animation( $name, $properties... ) {
    @keyframes #{$name} {
        @content;
    }
    animation-name: $name;
    animation: $properties;
}
.logo {
    @include animation( move, 5s ease ) {
        from { left: 0; }
        to { left:100%; }
    }
}
```

Now our mixin accepts a required parameter for the `animation-name` and then a variable number of `$properties` which can be used with the `animation` shorthand property.

Our mixin declaration is now much shorter and much more flexible. But we can take this one step further.

One of our goals as diligent Sass developers is to reduce repetition and thinking time wherever appropriate.

When creating an animation in this way, we're generating the keyframes and animation properties at the same time. So really, we don't need to

provide a meaningful name to the animation because Sass takes care of outputting everything for us.

Instead of having to think of an appropriate name for the animation, we could just have Sass generate a unique one for us automatically.

Sass has a built-in function called `unique-id()` which returns a randomly generated nine character string of alphanumeric characters

- just letters and numbers.

Using this unique reference, we can remove the `$name` parameter from the mixin declaration and just save the unique id into a variable.

This variable can then be used with interpolation to create a single animation shorthand comprised of the `$name` and other `$properties` in a single line.

```
@mixin animation( $args... ) {
    $name: unique-id();
    animation: #{$name $args};

    @keyframes #{$name} {
        @content;
    }
}

.logo {
    @include animation( 1s ease ) {
        from { left: 0; }
        to { left:100%; }
    }
}
```

The only downside to this method is when needing to create a set of keyframes that will be reused multiple times by multiple selectors

- with different values of `animation-delay` for example. With this mixin this isn't possible as the name of the block of keyframes is abstracted away as a randomly generated string.

I'd probably argue that we've refactored our mixin one step too far and actually made it overly complex, a little too abstract and not suitable for creating reusable animations.

```
@mixin animation( $name, $properties... ) {
    @keyframes #{$name} {
        @content;
    }
    animation-name: $name;
    animation: $properties;
}
```

```
.logo {
    @include animation( move, 5s ease ) {
        from { left: 0; }
        to { left:100%; }
    }
}
```

While we've learned a couple of advanced Sass tricks (which is always interesting of course), I'd revert our final code back to the previous iteration. Passing a name and a series of animation properties. That way, our mixin is still pretty lean but also a bit more meaningful and flexible enough if we want to reused our keyframes elsewhere.