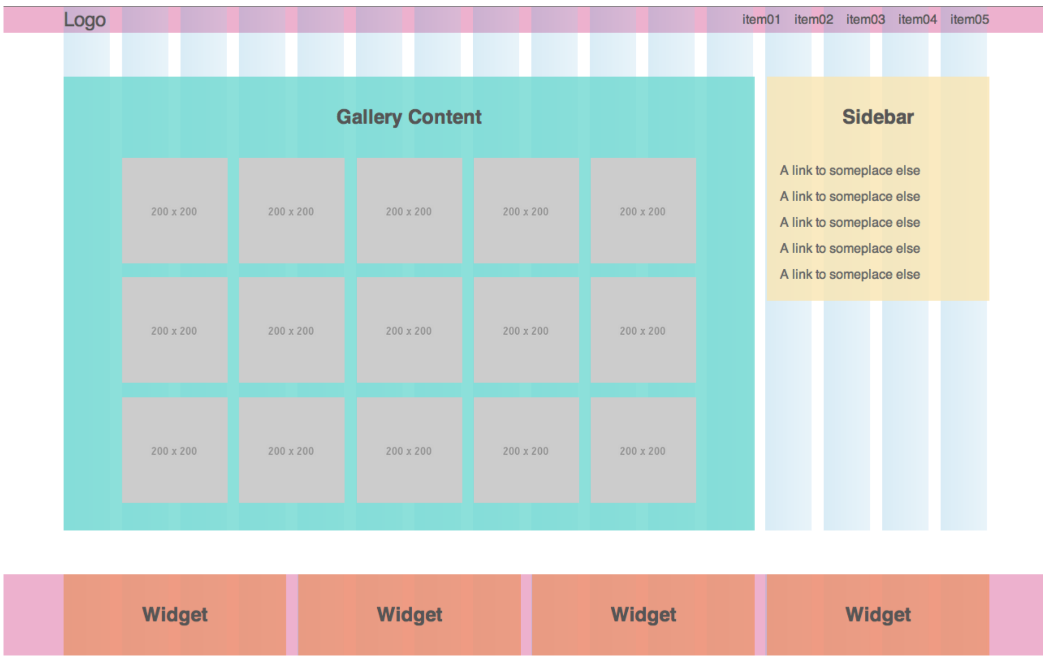
A More Complex Layout (Part 2)

We have completed a large part of the work in the previous chapter when we learned how to center `.gallery` within `.content`. We will finish the rest of the layout in this chapter.

You will learn:

- How to create a gallery with the `span()` mixin
- How to create a gallery with the `gallery()` mixin

Here's what we're building again:

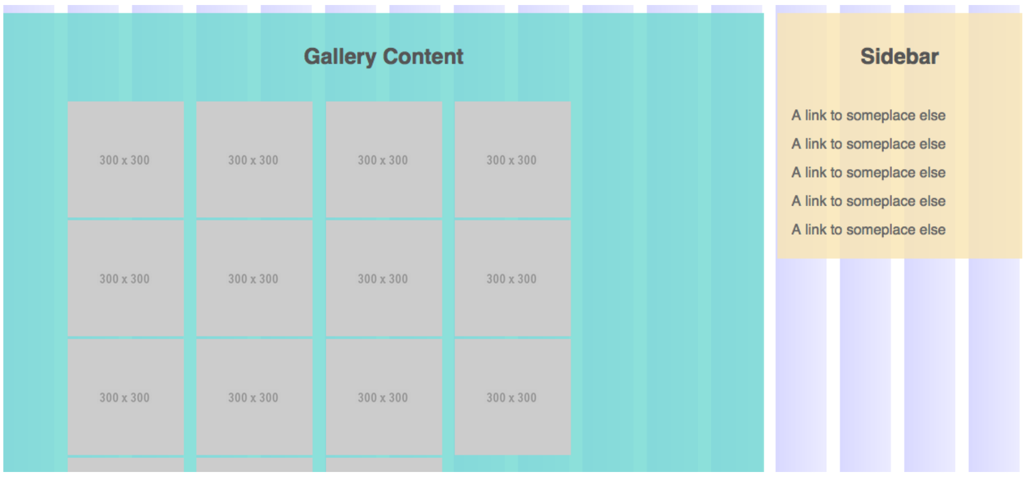


Building The Gallery Content

Once we have made sure that `.gallery` takes up 10 columns and is centered within content, we can proceed to build the gallery.

Each row in the gallery contains 5 `.gallery__item`s. This would mean that each `gallery__item` takes up 2 of 10 columns.

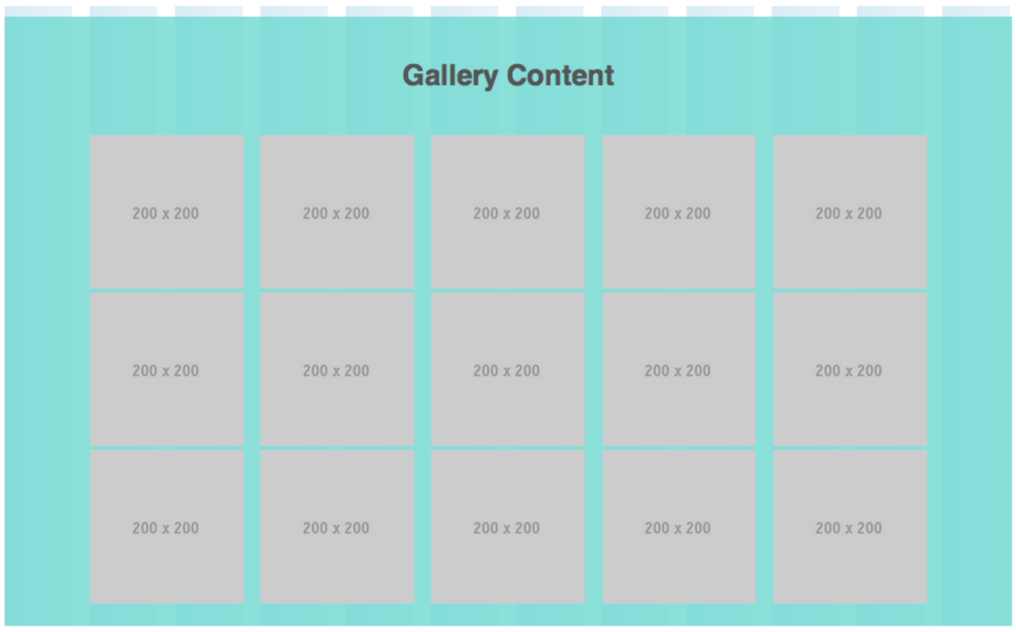
```
// Scss
.gallery__item {
  @include span( 2 of 10 );
}
```



The reason each row contains only 4 items and not 5 is because we have forgotten to remove the final margin on the right side of every 5th item. This is required because we are using the `after` gutter-position now.

To remove the `margin-right` of every 5th item, we need to use the `nth-child` pseudo class.

```
.gallery__item {
  @include span( 2 of 10 );
  &:nth-child(5n) {
    @include last;
  }
}
```



There’s just a tiny bit more to be done here. Notice that the space between the left and right of each `.gallery__item` is different from the space on the top and bottom. We can fix that easily with the `gutter` function since we know what context to use.

Uniform White Space

We have to supply `margin-bottom` with a percentage value that equals the gutter size to make the spaces between the gallery items uniform.

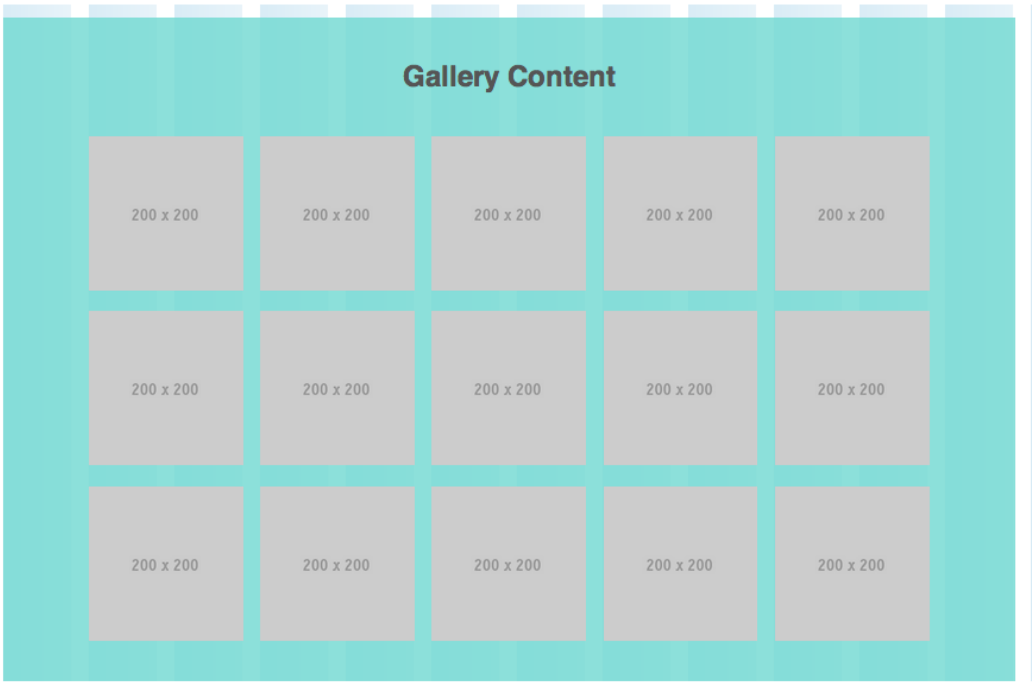
Since the `gutter()` function returns a percentage value for the horizontal space, we can use the same value for the vertical space.

This works because percentage values of `margin-top` and `margin-bottom` use the width of the container as 100%.

Since we know the context is 10 columns, the space for one gutter is `gutter(10)`.

```
// SCSS
.gallery__item {
  @include span( 2 of 10 );
  margin-bottom: gutter(10);
  &:nth-child(5n) {
    @include last;
  }
}
```

You should now have an evenly spaced gallery like the following:



View Source code

We have successfully created the `.gallery` component! Have you noticed that we kept using the context of `10` when working with this portion of the code?

We can make the code DRYer with the `nested()` mixin.

The `nested()` Mixin

The `nested()` mixin allows you to tell Susy to reuse a specific context for a code block. It the need for us to keep writing the nested context (10 in this case) in our functions and mixins.

Here's what the code looks like if we used the `nested()` mixin:

```
@include nested(10) {
  .gallery__item {
    @include span(2);
    margin-bottom: gutter();
    &:nth-child(5n) {
      @include last;
    }
  }
}
```

Notice how we took out the context of `10` from both the `span()` and `gutter()`. That's how you can use `nested()` easily.

View Source code

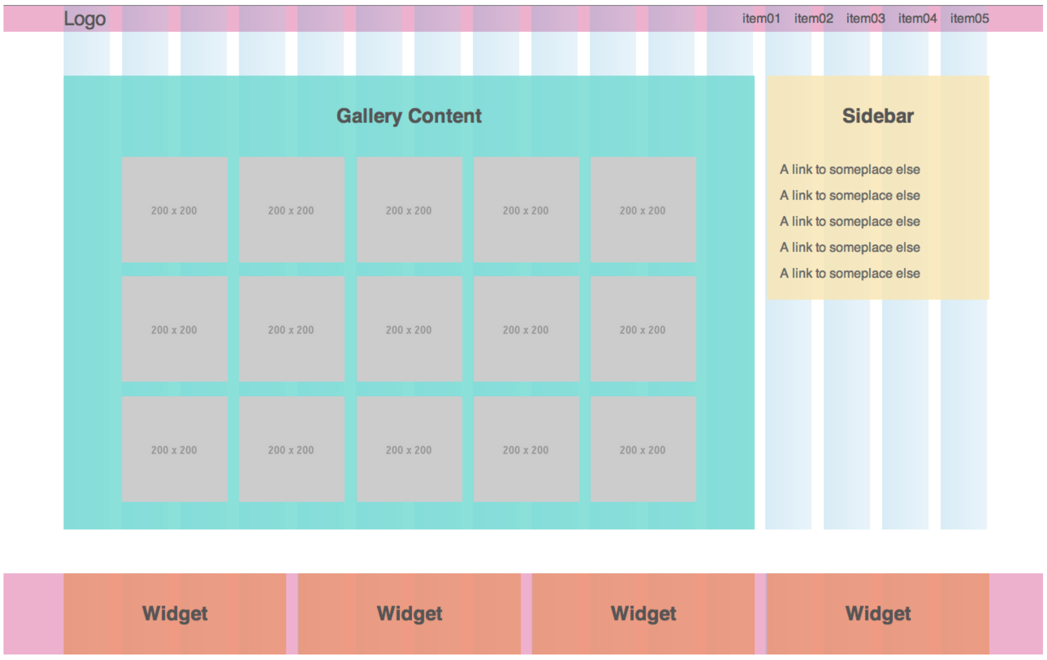
The `nested()` mixin is the mixin we will keep reusing to help us control the context without repeatedly writing ourselves. You'll see how it helps us out when we work on responsive websites.

Let's move along for now.

Now that we're done with `.content`, we can move on to the `.site-footer`.

The Footer

Let's remind ourselves what the footer looks like before we continue.



There is a pink background in the footer that spans the entire viewport. This means that the `.wrap` container must be placed within a `.site-footer`, just like the `.site-header`.

```
<!-- HTML -->
<footer class="site-footer">
  <div class="wrap"></div>
</footer>
```

There are 4 blocks of widgets in the footer section to simulate 4 different content blocks that might be placed on a footer in a working website.

```
<!-- HTML -->
<footer class="site-footer">
  <div class="wrap">
    <div class="widget"><h2>Widget</h2></div>
    <div class="widget"><h2>Widget</h2></div>
    <div class="widget"><h2>Widget</h2></div>
    <div class="widget"><h2>Widget</h2></div>
  </div>
</footer>
```

You may have noticed that these widgets can be styled in the same manner as the gallery. Since each `.widget` takes up 4 of 16 columns, the Sass would look like this if we follow the same route:

```
// SCSS
.widget {
  @include span(4 of 16);
  &:nth-child(4n) {
    @include last;
  }
}
```

Note: We don't have to add a clearfix to `.site-footer` because these widgets are contained within `.wrap`, which already has a clearfix.

Everything we have done up to this point now was the tedious way to get a gallery up and running. There is a much simpler way. Let me introduce you to the `gallery()` mixin.

The `gallery()` Mixin

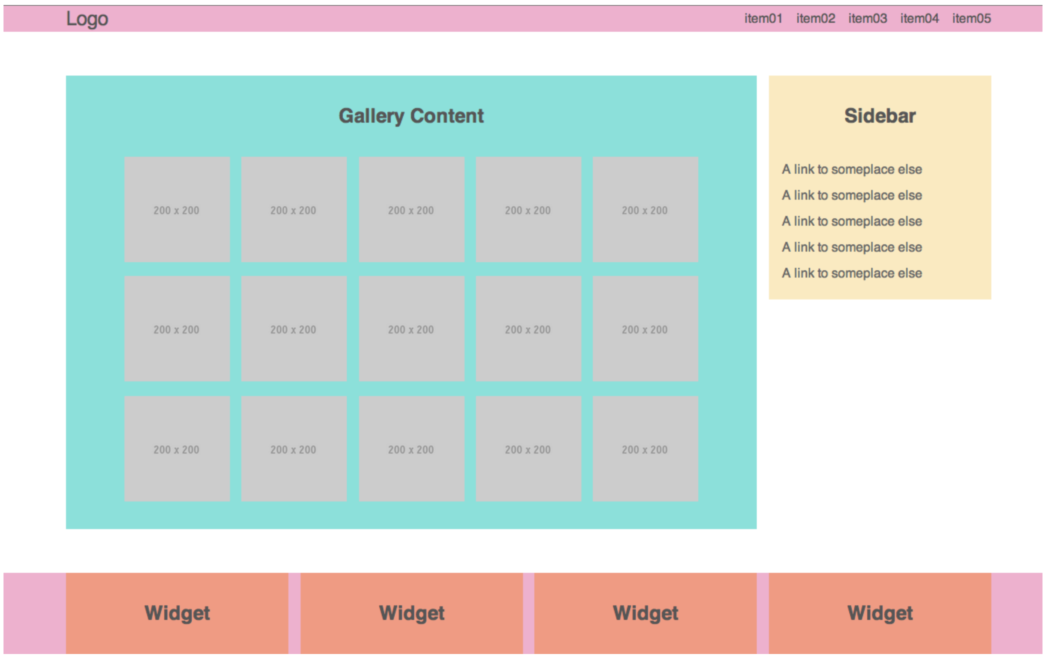
The `gallery()` mixin is specially created to make galleries with Susy. It takes in the same arguments as the `span()` mixin and can only create gallery items that take up the same number of columns.

```
// SCSS
.widget {
  @include gallery (<$span> of [<$context>]);
}
```

Since `.widget`s take up 4 of 16 columns each, that gives us:

```
// SCSS
.widget {
  @include gallery( 4 of 16 );
}
```

And we’re done!



Let’s explore the CSS output from this `gallery()` mixin to understand how it works.

```css
/* CSS */
.widget {
  width: 24.05063%;
  float: left;
}
.widget:nth-child(4n + 1) {
  margin-left: 0;
  margin-right: -100%;
  clear: both;
  margin-left: 0;
}
.widget:nth-child(4n + 2) {
  margin-left: 25.31646%;
  margin-right: -100%;
  clear: none;
}
.widget:nth-child(4n + 3) {
  margin-left: 50.63291%;
  margin-right: -100%;
  clear: none;
}
.widget:nth-child(4n + 4) {
  margin-left: 75.94937%;
  margin-right: -100%;
  clear: none;
}
```

There are huge differences between the output from the `gallery()` mixin and the `span()` mixin. Here are the differences:

1. Every item is floated to the left
2. Each item has its own value for `margin-left`
3. Each item has a `margin-right` of -100%
4. Only the first item is cleared, while the rest are not.

The `gallery` mixin creates such different CSS because it uses a technique called the Isolate Technique.

The Isolate Technique is a method that can be used to avoid subpixel rounding errors. This technique is slightly more advanced and will be covered in the later chapters after we learn how to make a responsive website with Susy.

A Quick Wrap Up

We have covered a great deal about Susy in these two chapters. Specifically, we covered these 4 things:

- the `span()` function,
- the `gutter()` function,
- the `$spread` argument and
- the `gallery()` mixin.

Each function or mixin performs a different role. Remember what they do because they will be of great benefit from here on.

In the next chapter, you will learn everything you need to know about media queries to build a responsive website.

Let’s move on.