



Learning Susy

How to Make Ultra Flexibile Layouts Easily
with the Susy Framework

By Zell Liew

Table of Contents

1. [Introduction](#)
2. [Creating Your First Project](#)
3. [The Scss Syntax](#)
4. [The Susy Map](#)
5. [Your First Layout](#)
6. [Susy Context](#)
7. [A More Complex Layout](#)
8. [A More Complex Layout \(Part 2\)](#)
9. [Media Queries](#)
10. [Breakpoints with Susy](#)
11. [Susy Background Grid](#)
12. [Building A Responsive Layout](#)
13. [Understanding Gutter Positions](#)
14. [The Isolate Technique](#)
15. [Asymmetric Grids with Susy](#)
16. [Asymmetric Grids With Susy \(Part 2\)](#)
17. [Static Grids With Susy](#)
18. [Susy Settings](#)
19. [Multiple Susy Grids](#)
20. [Susy Shorthand](#)
21. [Handling Difficult Susy Contexts](#)
22. [Integrating Susy With Your UI Kit](#)
23. [Sass Architecture](#)
24. [Wrapping Up The Book](#)

Introduction

Today's websites are much tougher to create compared to the past. Now, we need to create websites with layouts that can work with potentially an unlimited number of viewports on all types of screensizes.

A second problem we're facing in today's development world is that we need to develop fast. Really fast. Traditional layout grids help with the initial quick prototype, but are difficult to modify as most introduce a big bloated mess of code which, odds are, you don't even use 90% of.

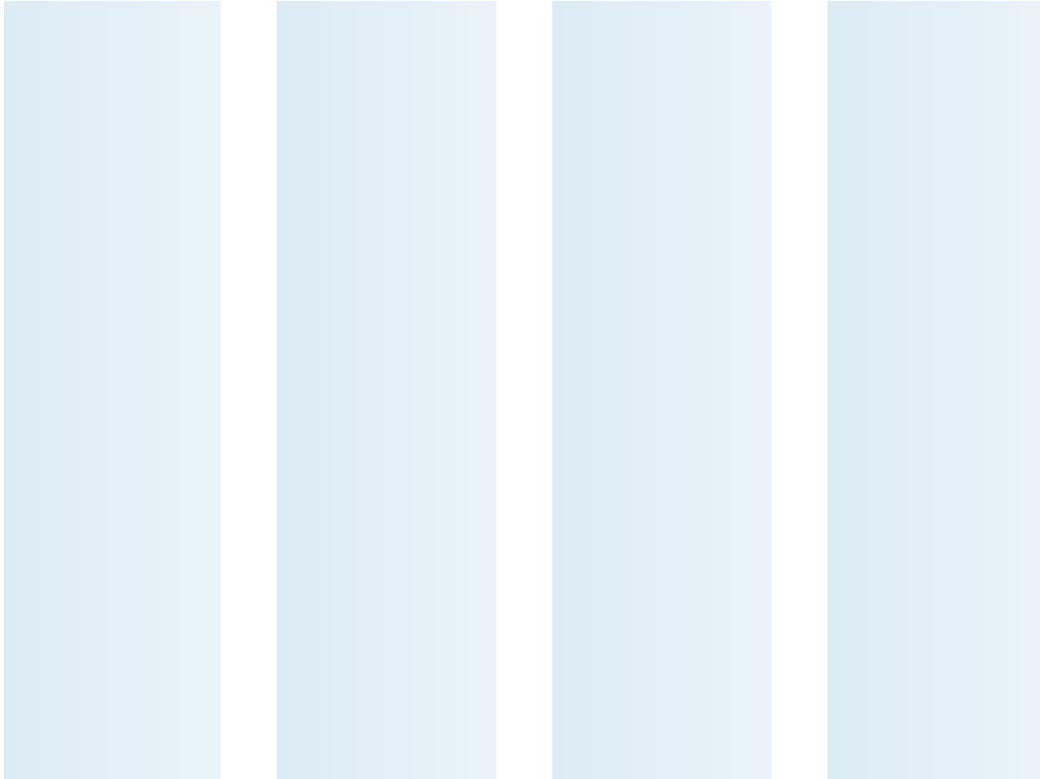
What we need is something light, flexible, easy to use, quick to change and to prototype.

Susy does all of that.

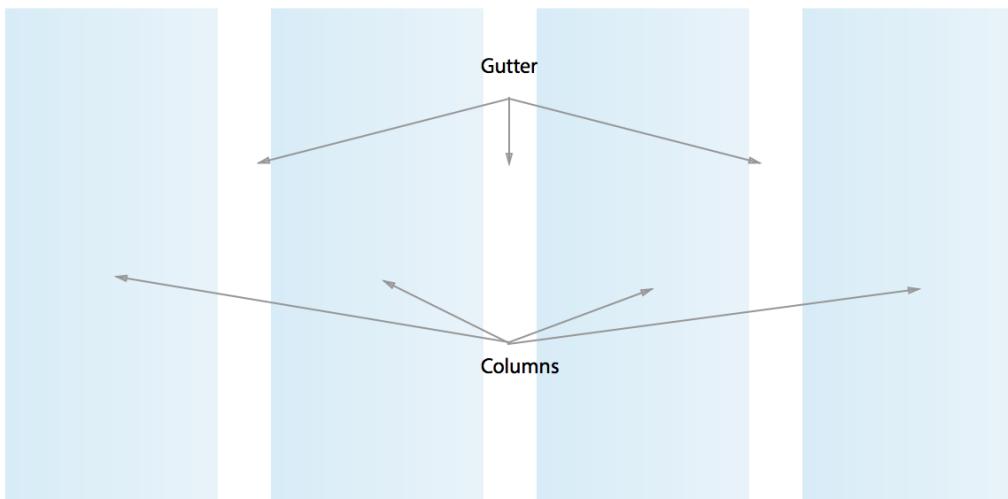
What is Susy?

In order to answer this question, we have to first answer "What is a grid?"

Grids in web terminology are nothing more than a set of vertical lines running from the top to the bottom of a page. They originate from print design, and are now used by web designers every day in their website designs to organize and present information in an orderly manner.

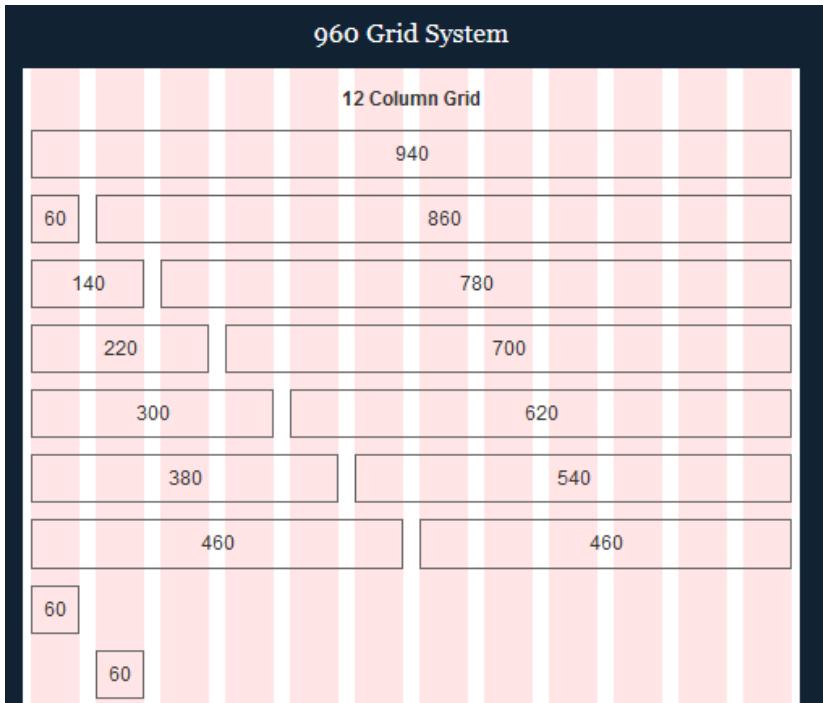


These vertical lines help to segment the page into two kinds of vertical spaces. We call the thicker space a column and the narrower space a gutter. The order and position of the elements on the web is known as a layout.

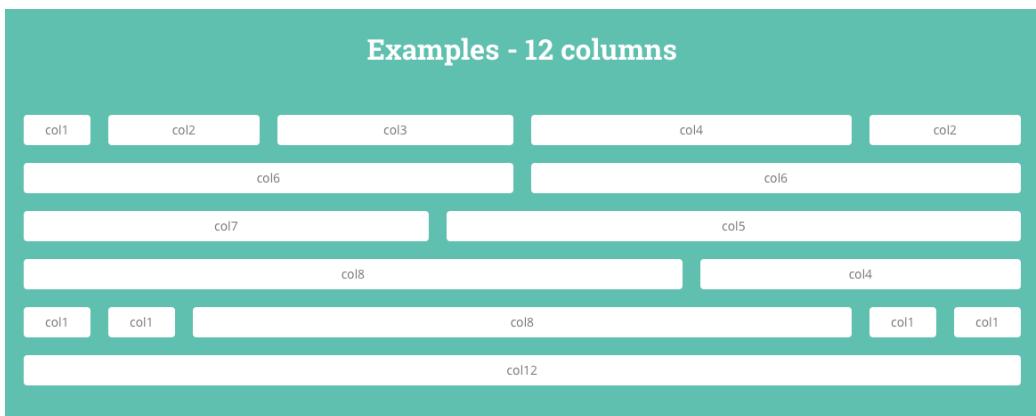


Grid systems have been around for quite a while, way before the web went mobile and responsive. You may even recognize some of these systems yourself!

The 960 grid system was one of the most widely used ones at one point. Every website used it as a base for their design. That was way back in 2010 when there was only one prevalent monitor size with a width of 1024px



Larger monitor sizes came out into the market (1280px) after that and everyone started switching to the 1140px grid system.



And as technology continued to improve and mobile devices gradually became the prevalent method for surfing the web, responsive grid frameworks such as Bootstrap and Foundation became a god-send. They were extremely useful for a long time. These frameworks catered to various device widths that were very popular at that time, like the iPhone 3, iPhone 4, iPad and popular desktop sizes like 1280px.

However, the infinite number of device types nowadays are starting to make even robust frameworks like Bootstrap and Foundation inadequate. We can no longer design for a few devices and hope that our website still responds nicely for everything else in between.

Since everyone has their own website these days, designers are challenged to come up with fresh and unique designs, some of which use grids with unequal widths.

Unfortunately, traditional grid frameworks like Bootstrap and Foundation lack the ability for you to customize your grids to that extent.

That is why Susy was born.

Susy is a layout engine. It provides you with the **tools to build your own grid framework** that matches the needs of your website. You have the flexibility and freedom to design however you like when you use Susy.

Why Susy?

So why should you take your precious time to learn Susy?

Susy is fundamentally different from anything you might have seen so far. It gives you the complete freedom to create anything. It allows you to express yourself and your design without being trapped by practices that have taken hold on the internet.

After developing a solid understanding of Susy, you will never ever have to refer back to the documentation while you are coding. Development time really speeds up after that.

The good thing about Susy is that it only requires you to use Sass, a great preprocessor that many in the industry are using. There are no external dependencies, which means that errors from dependencies and versioning are

kept to a minimum.

Why this book?

Susy makes things extremely simple for front-end developers by abstracting out a large chunk of CSS. Styling with CSS may appear simple on the surface, but when layouts don't work the way we expect them to, it can become difficult to find out what's wrong.

I've learned a lot from working exclusively with Susy for the past 6 months. It wasn't always easy to find answers to questions and I had to piece together answers from different sources. There were many times when I had to invent my own solutions to some of these problems.

I want to teach you how to code with Susy, and allow you to build the layout you always wanted, but found too difficult to do.

Who is This Book For?

My goal for this book is to make it simple and easy to understand. So you should be able to follow along nicely even if you've just started learning about Sass.

Sass has two different syntaxes for us to work with. For the duration of this book, I'm going to use the SCSS syntax because I'm personally more familiar with that.

I assume you have a basic working knowledge of Sass. You'll do fine if you had some experience compiling Sass code with the terminal or any preprocessor program.

How to Best Use This Book

If you're totally new to Susy, I would suggest you go through the book sequentially through the chapters. Each chapter builds on the lessons taught in the previous chapter and it might be confusing if you skip around.

While going through the book the first time, I suggest you manually type in the code into your code editor because it's we learn more effectively when we do so.

Getting In Touch

Feel free to email me at zellwk@gmail.com for any questions you might have regarding Susy or if you just want to say hello. I'll read and respond to every email.

Creating Your First Project

Starting your first project with Susy is almost the same as starting any project with Sass. The difference is that you'll need to add the Susy library to your Sass code.

There are lots of different ways that you can do this, depending on how you structure your Sass workflow. In this chapter, we will walkthrough the various methods on how to setup a basic Susy project and you can pick the one you feel most comfortable with.

We will create the same project structure and compile them using:

1. The Terminal
2. Compass
3. Codekit / Prepros
4. Grunt
5. Gulp

You should be able to compile Susy to Sass successfully by the end of this chapter.

Feel free to skip this chapter if you already have a Sass project working properly with Susy.

Note: If you are totally new to Sass and Susy, I suggest you read through the installation for the Terminal and Compass. Take a look at Codekit and Prepros too if you are uncomfortable with the command line. Finally, feel free to skip Grunt and Gulp.

Compiling With The Command Line

Let's begin with the bare bones method that uses the command line. Read through this section even if you don't feel comfortable with the command line because this is where we will set the foundation for the rest of the methods.

We need to make sure that both the Sass gem and the Susy gem are installed on your computer to use this method. For Mac users, you can install Sass and Susy with the following commands using the Terminal application:

```
# Command Line  
$ sudo gem install sass  
$ sudo gem install susy
```

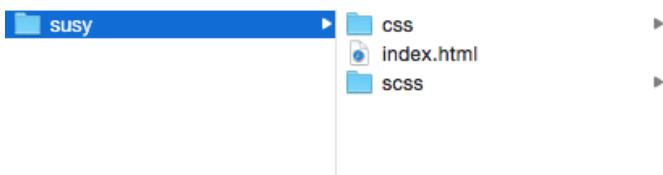
If you have previous versions of Sass and Susy installed, remove them with the `gem clean` command

```
# Command Line  
$ sudo gem clean
```

For Windows users, you must make sure you have [Ruby](#) installed on your system. You will also need to have Ruby Gems installed. Run the same code (without sudo) and you should have Susy and Sass installed.

Once the gems are installed, begin your project by adding an `index.html` file, a `css` folder and a `scss` folder.

The `css` folder holds the compiled css that the `index.html` reads while the `scss` folder holds all the Sass code for your project.



Place a `styles.scss` file within the `scss` folder and import Susy into your project.



```
// SCSS  
@import "susy";
```

When you're done, run the following command:

```
sass --watch scss:css -r susy
```

Sass should now watch for changes in any of your scss files and recompile whenever something is changed.

```
>>> Sass is watching for changes. Press Ctrl-C to stop.  
      write css/styles.css  
      write css/styles.css.map
```

Before we end this section off, I'm sure you already know about the importance of a reset file when working on frontend development. We're going to create this reset file and import it into our project.

To do so, first create a file named `_normalize.scss` and place it within your scss folder. Copy the normalize.css code from <http://necolas.github.io/normalize.css/> and paste it into your the file we just created.

Next, open up the styles.scss and write the following line:

```
@import "normalize";
```

This `@import` statement tells Sass that to look for the file `_normalize.scss`. Once found, insert its contents at the line that the `@import` statement is found.

We're done with creating a project with Sass and Susy.

[View Source Code](#)

Compiling With Compass

You may be familiar with Compass if you already know about Sass. If you want to create and run your project with Compass, be sure to install Compass (along with Sass and Susy) first.

```
# Command Line
$ sudo gem install sass
$ sudo gem install compass
$ sudo gem install susy
```

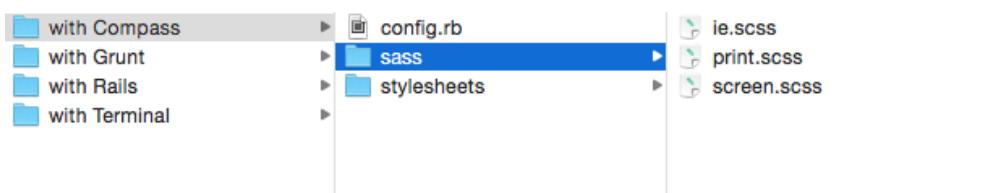
If you ran into any errors while compiling with Compass, I suggest you install the alpha version instead of the stable one. This would resolve most compilation errors.

```
# Command Line
$ sudo gem uninstall compass
$ sudo gem install compass --pre
```

You can initialize a Compass project by running the following command:

```
compass init
```

Compass will create a `config.rb` file along with a `sass` and a `stylesheet` folder for your sass and css files respectively.



Now, open up config.rb and you'll see a few commands. We have to tell Compass to use the Susy gem by requiring it with a line of code, like this:

```
1 require 'compass/import-once/activate'
2 # Require any additional compass plugins here.
3 require 'susy' | Insert this in
4
5 # Set this to the root of your project when deployed:
6 http_path = "/"
7 css_dir = "stylesheets"
8 sass_dir = "sass"
9 images_dir = "images"
10 javascripts_dir = "javascripts"
11
12 # You can select your preferred output style here (can be overridden via the command line):
13 # output_style = :expanded or :nested or :compact or :compressed
14
15 # To enable relative paths to assets via compass helper functions. Uncomment:
16 # relative_assets = true
17
18 # To disable debugging comments that display the original location of your selectors. Uncomment:
19 # line_comments = false
20
21
22 # If you prefer the indented syntax, you might want to regenerate this
23 # project again passing --syntax sass, or you can uncomment this:
24 # preferred_syntax = :sass
25 # and then run:
26 # sass-convert -R --from scss --to sass sass scss && rm -rf sass && mv scss sass
27
```

You can optionally change the file paths if you are more comfortable with `css` and `scss` folders instead of `stylesheets` and `sass` folders. Just be sure to change the folder names accordingly.

```
1 require 'compass/import-once/activate'
2 # Require any additional compass plugins here.
3 require 'susy'
4
5 # Set this to the root of your project when deployed:
6 http_path = "/"
7 css_dir = "css"
8 sass_dir = "scss" You can change these paths if you want to
9 images_dir = "images"
10 javascripts_dir = "js"
11
12 # You can select your preferred output style here (can be overridden via the command line):
13 # output_style = :expanded or :nested or :compact or :compressed
14
15 # To enable relative paths to assets via compass helper functions. Uncomment:
16 # relative_assets = true
17
18 # To disable debugging comments that display the original location of your selectors. Uncomment:
19 # line_comments = false
20
21
22 # If you prefer the indented syntax, you might want to regenerate this
23 # project again passing --syntax sass, or you can uncomment this:
24 # preferred_syntax = :sass
25 # and then run:
26 # sass-convert -R --from scss --to sass sass scss && rm -rf sass && mv scss sass
27
```

Once you're done with changing the file paths, run the `compass watch` command.

```
Zells-MacBook-Pro:with Compass zellwk$ compass watch
>>> Compass is watching for changes. Press Ctrl-C to Stop.
```

Of course, be sure to `@import susy` and `@import normalize` as we did above. That's all you need to set up a Susy project with Compass.

[View Source Code](#)

Compiling With Codekit / Prepros

[Codekit](#) is a tool for the Mac that helps with watching and compiling Sass to CSS (Codekit costs \$32).

If you are on Windows, [Prepros](#) would be the equivalent (Prepros costs \$29).

Setting up a project with Codekit or Prepros is the same as setting up a project with the command line as described above.

The difference is that once you complete the setup, you can drag the whole project into either Codekit or Prepros and it will help you compile Sass to CSS automatically. This means you won't have a headache with the terminal.

If you are new to Sass and you are cool with buying an app, I suggest you start with this method.

Compiling With Grunt

Grunt is a JavaScript task runner that helps to automate numerous tasks when developing websites and applications. The beauty of task runners is that you do the hard work of configuring it once and it will do most of the work thereafter with a simple command.

Note: This approach is not for beginners.

You can use the same basic project folder structure you used when you set up the project with the Terminal

Before you begin to use Grunt, make sure you have the following installed on your system:

1. [NodeJS](#)
2. [Grunt CLI](#)
3. [Bower](#)

Since we are using Grunt and Bower in this project, we can set the project up to easily add or manage both Node and Bower dependencies for the project.

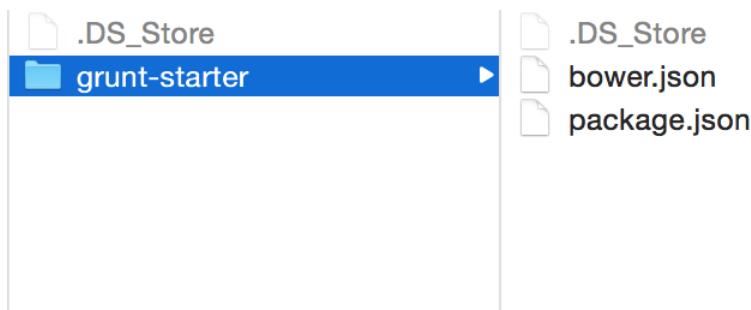
To do so, we require the `package.json` and `bower.json` files.

We can use the `npm init` command to create the `package.json` file and the `bower init` command to create the `bower.json` file.

```
# Command Line  
$ npm init
```

```
# Command Line  
$ bower init
```

These two files combined will allow you to easily add or manage dependencies in your project. Your folder structure should now be:



You will need two Grunt plugins installed into your project to convert Sass into CSS. In this project, we are going to setup LibSass to speed up the compilation as well.

You need to have two Grunt plugins installed into your project to convert Sass to CSS smoothly – `grunt-sass` and `grunt-contrib-watch`.

So install them both:

```
# Command Line
$ npm install grunt-sass --save-dev
$ npm install grunt-contrib-watch --save-dev
```

You will also need to install Susy as a Bower package.

```
# Command Line
$ bower install susy --save
```

Next, add the folders you have created in the command line section and within `styles.scss`, import Bower with this instead:

```
// Scss
@import "path-to-bower-components/susy/sass/susy";
```

In your project configuration for Grunt, setup Sass with the following format:

```
// Grunt-sass
sass: {
  app: {
    files: [
      expand: true,
      cwd: 'scss',
      src: ['*.scss'],
      dest: 'css',
      ext: '.css'
    ]
  },
  options: {
    sourceMap: true
  }
},
```

You'll also need to setup the watch task to make Grunt automatically recompile your Sass into CSS when any of your files change. You can also optionally set up Livereload (which will not be covered here)

```
watch: {
  sass: {
    files: ['scss/{,*/}*.{scss,sass}'],
    tasks: ['sass']
  },
  options: {
    livereload: true,
    spawn: false
  }
},
```

Finally, you will have to register a Grunt task in order to get Grunt working:

```
grunt.registerTask('default', ['sass', 'watch']);
```

Run this task by using the following command:

```
# Command Line  
$ grunt
```

[View Source Code](#)

Compiling with Gulp

Gulp is another JavaScript task runner that has been gaining popularity recently. It does the same things as Grunt, but is configured differently.

Note: This approach is not for beginners.

You can use the same basic project folder structure you used when you set up the project with the Terminal

Before you begin to use Gulp, make sure you have the following installed on your system:

1. [NodeJS](#)
2. [Gulp](#)
3. [Bower](#)

The setup is the same as compiling with Grunt. We have to create the

`package.json` and `bower.json` files with the `npm init` and `bower init` commands.

```
# Command Line  
$ npm init
```

```
# Command Line  
$ bower init
```

We have to install 3 packages with Gulp to compile Sass to CSS – `gulp` , `gulp-sass` .

```
# Command Line
$ npm install gulp --save-dev
$ npm install gulp-sass --save-dev
```

Since we're compiling Sass into CSS, we should also include a source map for debugging purposes. You have to install the `gulp-sourcemaps` package in order to use sourcemaps with Gulp.

```
# Command Line
$ npm install gulp-sourcemaps --save-dev
```

After which, you'll have to install Susy with Bower and import it.

```
# Command Line
$ bower install susy --save
```

```
// Scss
@import "path-to-bower-components/susy/sass/susy";
```

Next, we'll have to create a `gulpfile.js` and place it in the root of the project. Within this `gulpfile.js`, we will create the `styles` task to convert Sass into CSS.

```
var gulp = require('gulp');
var sass = require('gulp-sass');
var sourcemaps = require('gulp-sourcemaps');

// styles task
gulp.task('styles', function() {
  gulp.src('./scss/**/*.{scss,sass}')
    // Initializes sourcemaps
    .pipe(sourcemaps.init())
    .pipe(sass({
      errLogToConsole: true
    }))
    // Writes sourcemaps into the CSS file
    .pipe(sourcemaps.write())
    .pipe(gulp.dest('./css'));
});
```

We'll also watch the `scss` folder for changes and recompile Sass to CSS as necessary. At the same time, we can create a task to run in the command line.

```
gulp.task('default', ['sass'], function() {
  gulp.watch('./scss/**/*.{scss,sass}', ['sass']);
});
```

Run this task with the `gulp` command.

```
# Command Line
$ gulp
```

[View Source Code](#)

A Quick Wrap Up

You can set up the project to run with different compilers. Although each compiler is configured in a slightly different way, the project structure for a Sass project remains the same throughout.

Once you get a solid grasp of the project structure, you can work with any compiler you want.

Now that we have set up the project properly, let's move into the next chapter and find out about Scss, the CSS-like syntax for Sass that we will be using for the rest of the book.

The Scss Syntax

The code samples in this book are written with Sass, more specifically, with the Scss Syntax. If you are already familiar with CSS, you shouldn't have any problems comprehending the Scss syntax.

Within this chapter, you'll learn

- What is Sass
- What is Scss
- How to use Mixins in Scss
- How to use Functions in Scss
- How to compile Scss into CSS

Sass

CSS has been criticized by many programmers for many reasons such as

- There's no logic
- You cannot do nesting
- You cannot use functions
- You cannot use variables
- It can be extremely repetitive
- ... and the list goes on.

It becomes more difficult to code with pure CSS as websites gets significantly more complicated. To address these problems, CSS preprocessors began appearing in the web world.

CSS Preprocessors are called preprocessors because they are written in a slightly different syntax and are compiled into CSS.

This slightly different syntax enables developers to introduce logic into CSS, and allows the use of complication functions and calculations that are similar to regular programming languages.

The two most popular CSS preprocessors now are – [LESS](#) and [SASS](#), and both are used by veterans in the web industry. Susy only works with Sass, and that's why we're using Sass.

SCSS

Sass has two different formats. Sass and Scss.

Sass is written in a way that resembles Python and Ruby, where there are no curly braces like `{` and `}` that form the start and end of a selector, and there are no `;` at the end of every property.

Scss on the other hand resembles CSS entirely with the `{`, `}` and `;`.

Here's an example of the two syntaxes

```
// Sass
.test
background: red
```

```
// Scss
.test {
background: red;
}
```

Both of them compile into the same CSS

```
/* CSS */
.test {
  background: red;
}
```

We will use Scss because its similar to CSS. It's much easier for you to pick up Scss if you haven't used any preprocessors before. From this point onwards, Sass or Scss both refer to Sass in the Scss syntax.

Let's have a look at the reasons preprocessors are preferred and how to use them.

Using Variables

Variables reduce the need for you to repeat the same properties across different areas. The reason why you should use them is self explanatory once you know how to use them. Let's have a look.

Variables are declared with a `$` in front of the variable name.

```
$variable: 20px;
```

From here on, you can use `$variable` anywhere in your SCSS code and its value will be compiled to `20px`.

```
// SCSS
.test {
  font-size: $variable;
}
```

This outputs as

```
/* CSS */
.test {
  font-size: 20px;
}
```

You can store any value within variables. You can even store font families and other strings, numbers or even colors that you would like to repeat throughout the website. Very useful stuff.

Next, let's have a look at mixins.

Using Mixins

Mixins are shortcuts that help you write multiple CSS properties at once. They are a convenient way to allow you to write less, and yet, output the properties you need to CSS.

Mixins are defined with the `@mixin` key, followed by the mixin name and any arguments.

```
// Defining a mixin (SCSS)
@mixin color($color) {
  color: $color
}
```

Seasoned programmers will notice the resemblance to regular programming languages. `$color` is a variable that is defined for this particular mixin, and can be used within the mixin itself.

We won't be creating many mixins in this book, but we are going to use them. So the key focus here is actually knowing how to use mixins.

You can use mixins by adding `@include` followed by the mixin name and any arguments it may hold. Some mixins can be used without selectors while others have to be used within selectors.

```
// Using a mixin (SCSS)
.test {
  @include color(red);
}
```

```
/* CSS */
.test {
  color: red;
}
```

Mixins are often used to make more complex properties that require the output of more than one property. They can also contain functions and logic to output certain properties when a condition holds true, which allows for very complex programming to take place.

Susy provides us with a lot of mixins that are already preconfigured and ready for use. We will go into the details of these mixins and explain their output as we go through each section of the book.

Let's look at functions next.

Using Functions

Sass provides us with functions that we can use to return a specific value. Functions help us do complex calculations and returns a value that we can use somewhere else within the Sass code.

Functions are defined with the `@function` key, followed by the function name and any arguments required.

```
// Defining a function (SCSS)
@function multiply-by-two($number) {
  @return $number * 2;
}
```

As with mixins, we won't be defining many functions because Susy already has them defined for us. We just have to know how to use them.

You use functions by writing the function name along with its arguments like

```
multiply-by-two(20px);
```

```
// Using a function
.test {
  width: multiply-by-two(20px)
}
```

```
.test {
  width: 40px;
}
```

There are other ways of using functions, but in this book, we are only using functions in this way.

More about Sass

That's all you need to know about Sass and the Scss syntax to move forward with the book.

There's more to Sass than what I have explained here. I highly recommend you check out the Sass tutorials on [Level up tuts](#). It will help you do much more in future.

The Susy Map

Every Susy project begins with the `$susy` map. It is a set of instructions that Susy will use whenever you want it to create a grid for you. If you change the settings within the map, you change the instructions for Susy and it will create a different grid.

One of the keys to using Susy is to make sure you know how to use the Susy map and to understand its intricacies. We will stick with most of the defaults that Susy comes with to start off as they can be confusing for beginners.

In this chapter, you will learn:

- What is a map
- How to write the Susy Map
- How to use `columns`, `debug` and `global-box-sizing` keys

Let's take a look at how to use the Susy map.

The Map Object

Susy uses a Sass map to store the settings for your project. Sass maps are key and value pairs that contain information. They are similar to JSON objects and have the following syntax:

```
// Scss
$map :(
  key : value,
  key2: value2,
);
```

Each Sass map can hold an infinite number of `key` and `value` pairs. The information stored within each `key` is called the `value`. Each `value` can

be any type of information, ranging from `integer`, `string`, `list` or even another `map`.

Let me use an analogy if that came across as being too geeky.

Imagine you are looking for the definition of the word “dream” in a dictionary. You open up the dictionary and head towards the “D” section. And you found the word “Dream”. One of the meanings you find for “Dream” is “a series of thoughts, images, and sensations occurring in a person’s mind during sleep”.

A Sass Map is like a dictionary. It has the same structure. If we put this dictionary into a Sass map, this is how it would look like:

```
$dictionary: (
  "Dream" : "a series of thoughts, images, and sensations
  occurring in a person's mind during sleep",
  "another-word" : "Meaning for the word"
);
```

The `key` in this example is Dream. You use the key to find its value, which in this case, is “a series of thoughts, images, and sensations occurring in a person’s mind during sleep”.

There are other ways to use the map. You can even use it to store the color of fruits:

```
$color-of-fruit: (
  apple: red,
  banana: yellow,
  pear: green
)
```

You can search for the color of `pear` in this map and it will return `green`.

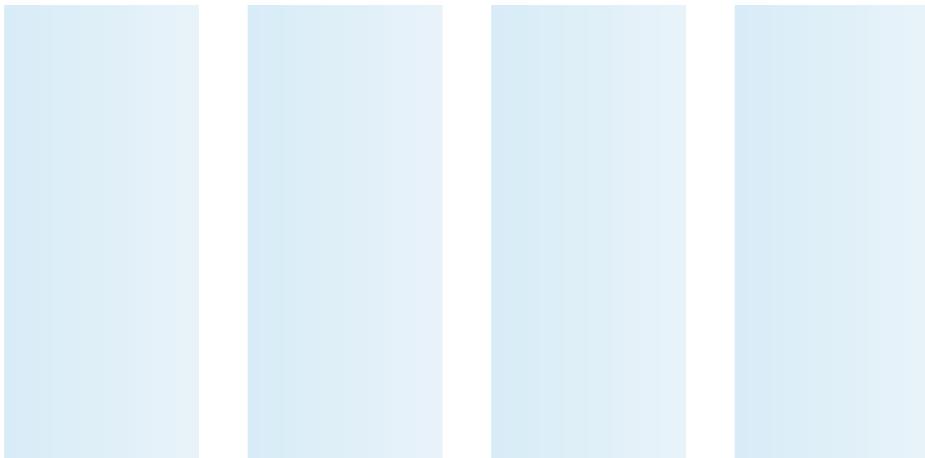
The good news is, you don’t have to know how to get the values out of the `$susy` map. You just have to know how to write them in.

Let's give the `$susy` map a try.

Configuring the Susy Map

Susy comes with some default settings set in the `$susy` map. You'll have to change its settings if you wanted to create a customized grid.

Say you want to create this grid now:



You can see that there are 4 columns in the grid (the 4 light blue columns in the background). To build this grid, We have to tell Susy that there are 4 columns in the grid.

We do so by using the `columns` key. It tells Susy the number of columns you'll be using for the grid.

```
$susy: (  
  columns: 4  
) ;
```

The light blue columns are created by the Susy debug helper so that you can see the grid. These columns are hidden by default and must be turned on with the `debug` key.

```
$susy : (  
  columns: 4,  
  debug: (image: show)  
);
```

`debug` is a special key within Susy that is meant for showing the helper background grid. It is written in a different way compared to `columns` because there are other parameters that you can play with within the `debug` key. We will explore them in a later chapter.

There is one more thing that I would like for you to add to this `$susy` map. That is to set `global-box-sizing` to `border-box`.

```
$susy: (  
  columns: 4,  
  global-box-sizing: border-box,  
  debug: (image: show)  
);
```

`global-box-sizing` tells Susy which box model to use for all the grid styles that Susy creates. We are changing it to `border-box` because it makes calculating layouts with CSS much easier. It is also the preferred setting that many web experts use.

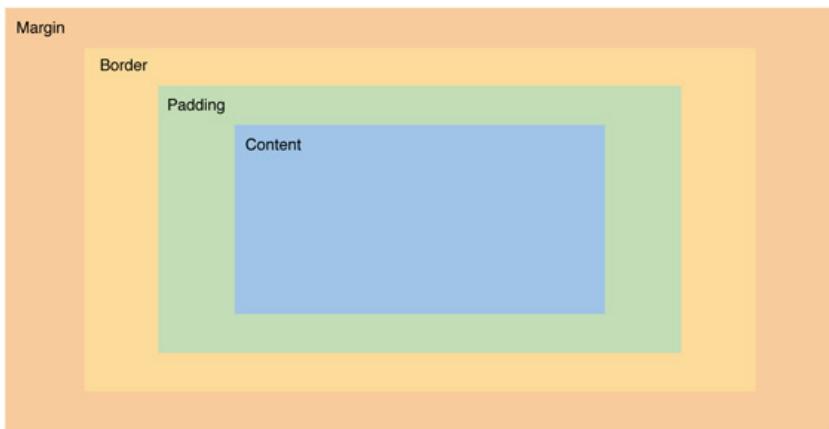
We have to tell the browsers to switch to the `border-box` sizing property by adding the `border-box-sizing()` mixin after the `$susy` map.

```
$susy: (  
  columns: 4,  
  global-box-sizing: border-box,  
  debug: (image: show)  
);  
  
@include border-box-sizing;
```

The box model is important to understand when working with CSS. It affects how you write CSS code greatly. Let's explore the box model before continuing.

Box Model

First, we have to know that every HTML element is a box that contains 4 different layers. These layers are the actual content layer, the padding layer, the border layer and the margin layer.

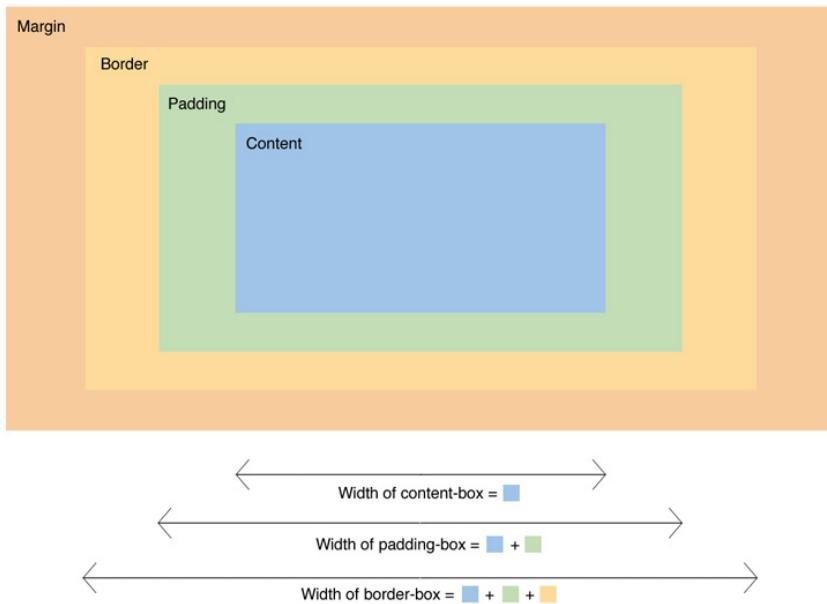


When there are this many layers, calculating the height and width of the HTML element becomes ambiguous. The `box-sizing` property is a cue for browsers to know what layers to include when measuring the height and width of the element.

There are three valid values for the `box-sizing` property:

1. `content-box`
2. `padding-box`
3. `border-box`

These values affect how browsers calculate the width and height of an element.



`content-box` is the default box model property given to all HTML elements. It tells browsers that the CSS `width` or `height` property refers only to the content section.

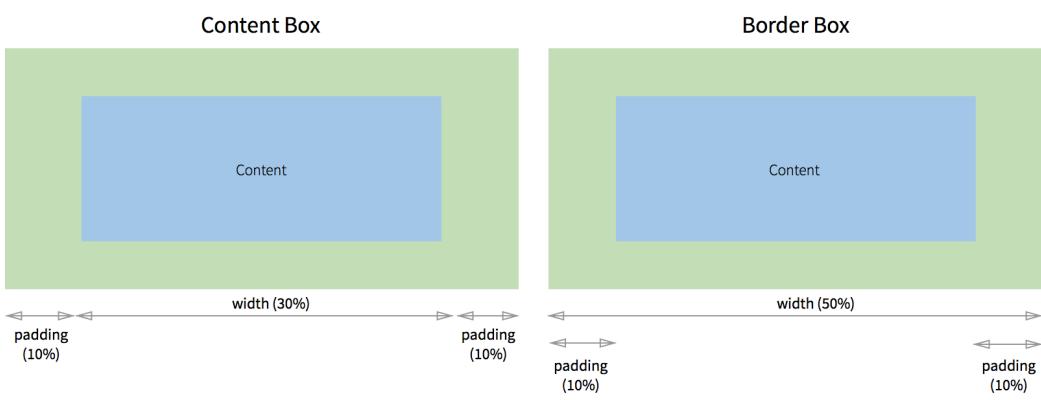
`padding-box` tells browsers that the CSS `width` or `height` property refers to the content section plus paddings. It's not recommended to use `padding-box` because browser support for it is not great.

`border-box` tells browsers that the CSS `width` or `height` property for the element is the addition of all paddings, borders and the content section.

`border-box` is the preferred box-sizing property of the three because it's more intuitive to think that the width of an element stretches from one of its borders to another.

Think of the last time you had to give an element some breathing space. The first instinct will probably be to add some `padding` to the element. We can put this into a more specific example:

Say you wanted to create an element that takes up 50% of the browser width and have a 10% breathing space between its contents and its edges.



This is what the code looks like:

```
/* CSS */
.content-box {
    /* Width plus padding equals to desired width (50%) */
    width: 30%;
    padding: 0 10%;
}

.border-box {
    /* Desired width is just width. Padding pushes content inwards */
    width: 50%;
    padding: 0 10%;
}
```

It's much easier to use `border-box` since you can write `width: 50%` directly instead of accounting for the breathing space like you'll do with `content-box`.

The `@include border-box-sizing` mixin provided by Susy adds border-box sizing to all HTML elements present on the webpage. This is the CSS it outputs:

```
/* CSS */
*, *::before, *::after {
  -moz-box-sizing: border-box;
  -webkit-box-sizing: border-box;
  box-sizing: border-box;
}
```

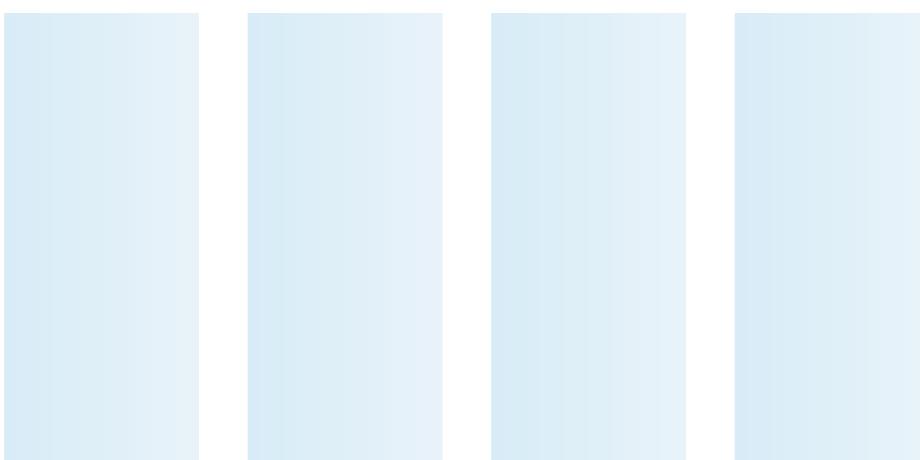
Using the Susy Map

The Susy map is automatically used whenever you use any Susy related mixins or functions. Let's add a `container` mixin to our Sass file and look at the results of this chapter:

```
<!-- html -->
<div class="wrap"></div>
```

```
// Scss
.wrap {
  @include container();
  height: 100vh; // This forces .wrap to 100% of your viewport
height, allowing you to see the background grid
}
```

You will now see the grid background show up on your screen.



[View Source Code](#)

A Quick Wrap Up

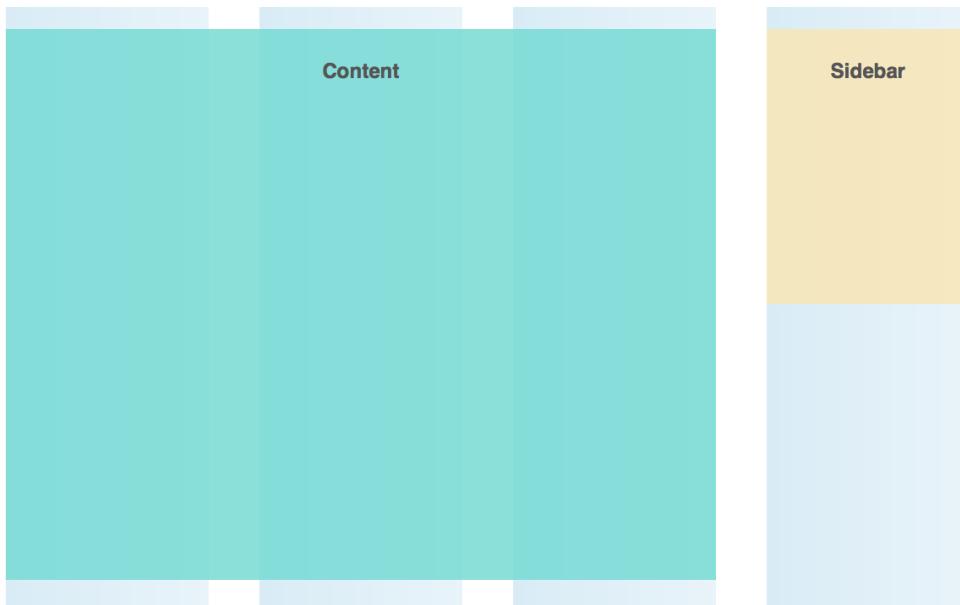
We have just covered the basics of Susy settings. You will need to know how to do this for every new grid that you create.

In the next chapter, You will learn how to create content areas and fit them onto the grid.

Your First Layout

We spent the last few chapters laying out the foundations and ensuring the project settings are correct. It's now time to create layouts with Susy.

Let's begin with something simple for your first layout:



Two mixins will be used to create this layout with Susy. We will be going through how to use them in this chapter.

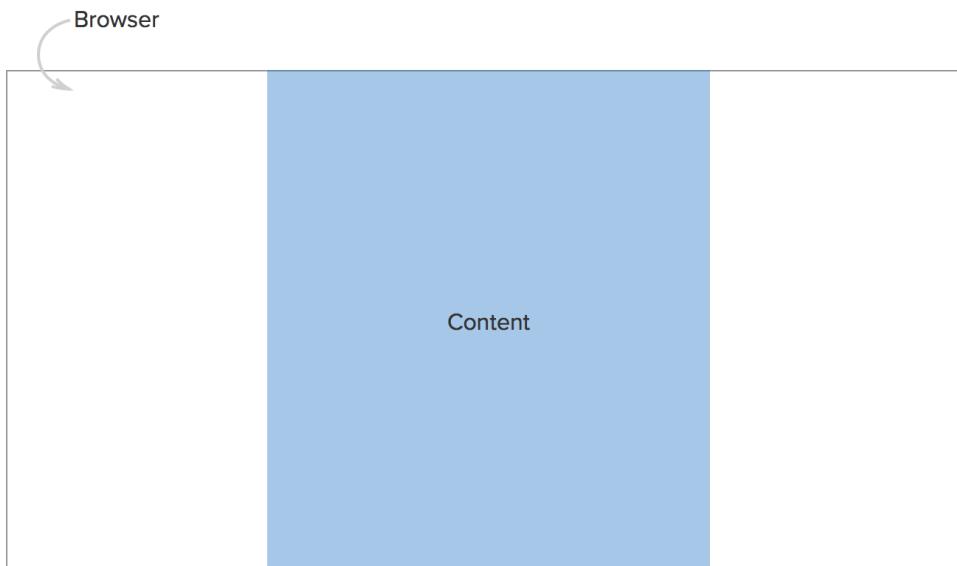
You will learn:

- How to use the `container()` mixin
- How to use the `span()` mixin

The `container()` mixin

Every grid requires a grid container. You can give this container any class you like. Common examples are `.page-wrap` and `container`. Let's keep it simple and give it a `.wrap` class.

The container does not take up the full width of the browser most of the time. It is common to center the container in the middle of the webpage when this happens.



To center the container, we need 3 properties: `width` , `margin-left` and `margin-right` .

```
/* Css */  
.wrap {  
    width: 960px;  
    margin-left: auto;  
    margin-right: auto;  
}
```

If the website is to be responsive, the container needs a `max-width` property instead of a `width` property.

```
/* Css */
.wrap {
  max-width: 960px;
  margin-left: auto;
  margin-right: auto;
}
```

Susy will create these required properties along with other properties when we use the container mixin.

```
// Scss
.wrap {
  @include container();
}
```

```
/* Css */
.wrap {
  max-width: 100%;
  margin-left: auto;
  margin-right: auto;
  // Other properties...
}
```

Say we want the maximum width of the container to be 1140px. We can tell Susy to create this container by adding the `container` key to the `$susy` map.

```
$susy:(
  columns: 4,
  container: 1140px,
  debug: (image: show),
  global-box-sizing: border-box
);
```

Susy will set the container to 1140px automatically. You can also use other units like em and rem if you prefer to.

```
.wrap {  
  max-width: 1140px;  
  margin-left: auto;  
  margin-right: auto;  
  // Other properties...  
}
```

[View Source Code](#)

Note: It is recommended to use a `<div>` as the Susy container. Don't use the `<body>` element because we need to give a `margin-left` and `margin-right` to the container, and `<body>` elements don't work well when margins are given.

Susy not only centers the content for us with the container mixin. It creates two additional sets of helpful properties.

The first set adds a `clearfix` to the container.

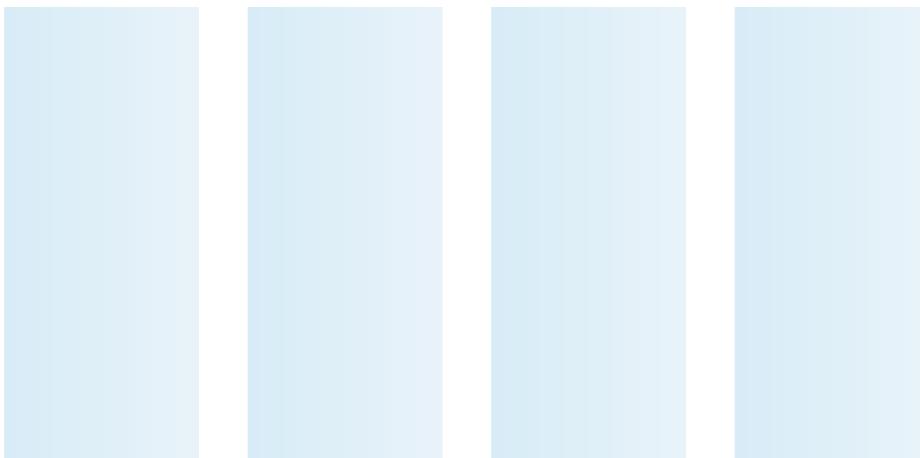
```
/* Css */  
.wrap:after {  
  content: " ";  
  display: block;  
  clear: both;  
}
```

A clearfix ensures that the container will not collapse when all its children are floated. Clearfixes are important when working with Susy because most of Susy's grid layout methods use floats.

The other set of properties help produce the 4 column background grid you saw in the previous chapter:

```
/* Css */
.wrap {
  background-image: linear-gradient(to right, rgba(102, 102,
255, 0.25), rgba(179, 179, 255, 0.25) 80%, transparent 80%);
  background-size: 26.31579%;
  background-origin: content-box;
  background-clip: content-box;
  background-position: left top;
}
```

This code is only produced if the `image` key within the `debug` key is set to `show`, just like what was set for our `$susy` map.



Once the container is up, we can start creating the grid layouts. Before that, let's write some CSS that will help us visualize these grids.

Taking Care Of CSS

HTML elements are iffy things. You cannot see them normally even though you know that they're there. However, we need to see these elements to ensure our layout is correct. To see these layouts, we will need to give each of the HTML elements a background color, plus a few extra CSS properties.

Since the book is really about Susy and not about these extra CSS properties, there is a “**Taking care of CSS**” section in every chapter to help speed things

up. This section provides you with the basic styles you need to be able to see these elements and backgrounds, and know whether Susy's grids have been positioned correctly.

I have opted to use the `vh` or viewport-height CSS3 property to create height and margins for demos. These heights are only used for demo purposes.

Note that you should never use a fixed height for your content (unless you know what you are doing).

Go ahead and paste the CSS for this chapter into your Sass file.

```
.content {  
  margin-top: 10vh;  
  height: 80vh;  
  background: rgba(113, 218, 210, 0.8);  
}  
  
.sidebar {  
  margin-top: 10vh;  
  height: 40vh;  
  background: rgba(250, 231, 179, 0.8);  
}  
  
h2 {  
  padding: 1rem 0;  
  text-align: center;  
  color: #555;  
}
```

Writing the HTML

Since we are taking advantage of the above CSS to speed up your learning, you have to be very careful and make sure you follow the classes I mention within this section or the styles won't apply properly.

Just in case you are new to HTML and CSS, I'll explain why I opted to structure the HTML this way to help you understand how to structure your own HTML.

The HTML for this chapter is:

```
<!-- HTML -->
<div class="wrap">
  <main class="content"><h2>Content</h2></main>
  <aside class="sidebar"><h2>Sidebar</h2></aside>
</div>
```

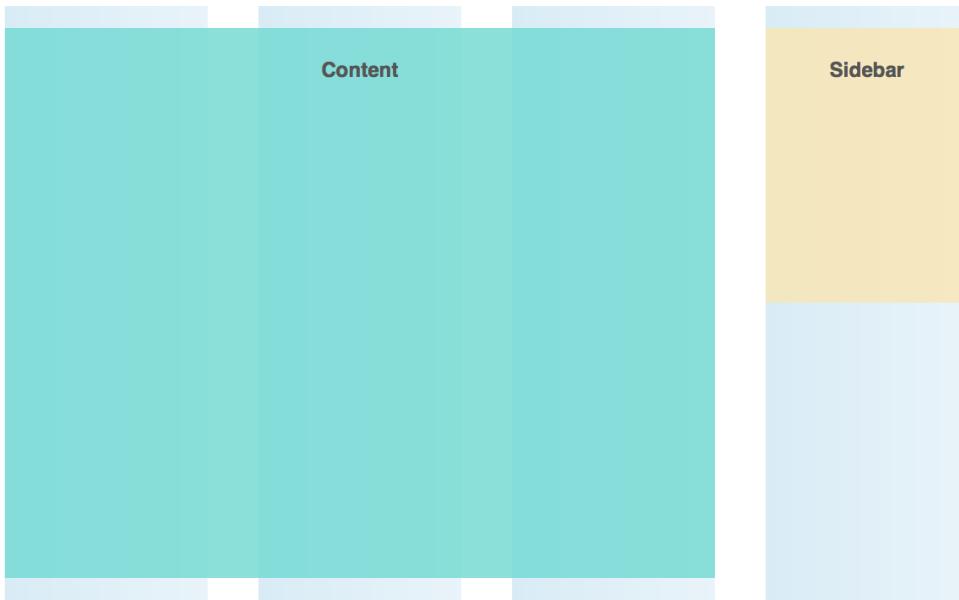
Classes used in this chapter are `.content` and `.sidebar` because it's easier to visualize and understand them. The `<h2>` tag is used within each `<div>` to act as the header element for the section. Some styles are also applied to `<h2>` so it looks solid.

[View Source Code](#)

Let's start making the layout with Susy.

Laying it out with Susy

We went through a lot before we got to this point. Let's take a breather and recall the layout we are trying to create for this chapter:



You only have to count the number of columns when working with Susy. In this layout, we can clearly see that `.content` takes up 3 of the 4 available columns and the `.sidebar` takes up the final 1 column. We'll use this information in the most important mixin you'll use in Susy: the `span()` mixin.

The Span Mixin

The `span()` mixin is used everywhere when you use Susy. The simplest way to use the span mixin is with this syntax:

```
// Scss
@include span( <$span> of [<$context>] [<$last>] );
// Note: Arguments within square brackets are optional arguments
```

`$span` determines the number of columns the element is going to take up.
`$span` will be 3 for `.content`.

`$context` is the total number of columns in the parent element. `$context` should be 4 since there are a total of 4 columns in `.wrap`. If this is left blank, Susy will obtain the `$context` argument from the `$susy` map, which

defaults to 4 in this case. Context is incredibly important to understand in Susy and we will explore more about this in the next chapter.

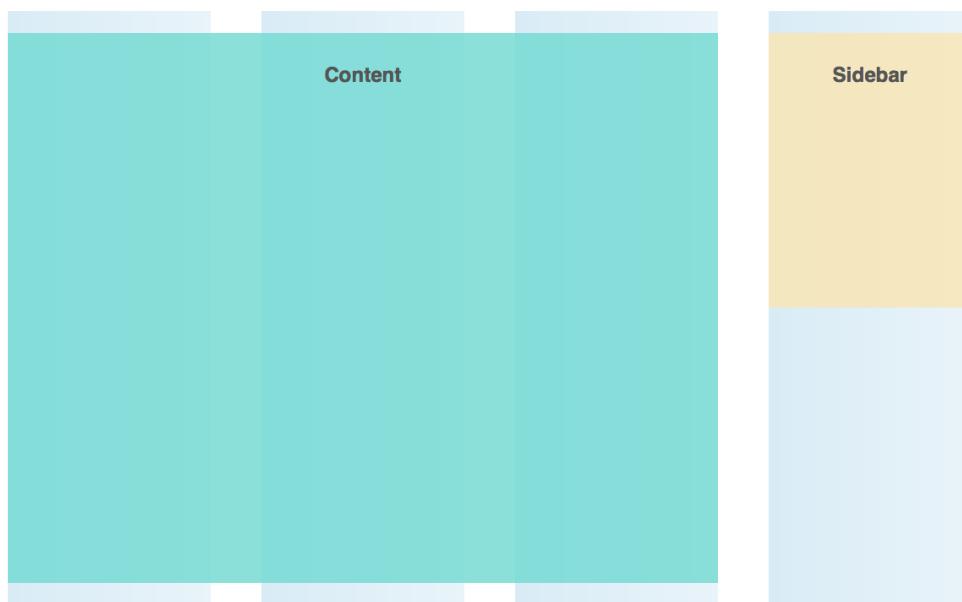
`$last` is an optional argument that tells Susy if this element is the last item in the row. The `last` keyword has to be supplied here for the last item on the row. We have to apply the `last` keyword for `.sidebar`.

We know that `.content` takes up 3 of 4 columns, `sidebar` takes up the final column of the 4 columns. `sidebar` is also the last item in the row. This will translate into:

```
// SCSS
.content {
  @include span(3 of 4);
}

.sidebar {
  @include span(1 of 4 last);
}
```

Susy will then work its magic and we get this:



[View Source Code](#)

Breaking Down The `span()` Mixin

Too much magic seems to be happening with just these two sentences. We have to understand the sorcery behind Susy if we want to fully understand it and wield its powers properly. Here we go:

Susy will output 3 properties whenever you use the `span()` mixin. The value of these properties will depend on the arguments given to the `span()` mixin and the settings in the `$susy` map.

These three properties are:

- `width` of the element (in %)
- `float` of the element (`left` or `right`)
- `margin` or `padding` of the element

The CSS output created by Susy for this chapter are:

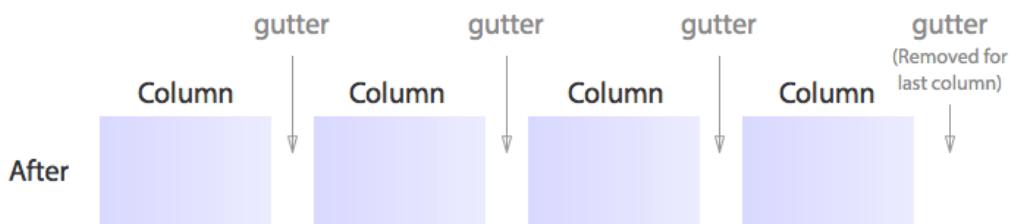
```
.content {  
  width: 73.68421%;  
  float: left;  
  margin-right: 5.26316%;  
}  
  
.sidebar {  
  width: 21.05263%;  
  float: right;  
}
```

Bear in mind that the value of these properties (and the properties produced) will change slightly depending on your `$susy` map settings. These properties are affected the most if you change the `gutter-position` setting on the `$susy` map.

Since we are using most of the default settings, `gutter-position` on the `$susy` map is set to `after`.

When using the `after` setting for `gutter-position`, Susy will create the gutter as a margin after every column. You will have to remove the margin of the last column of each row with the `last` keyword like what we did above.

Here's an illustration to make it simpler to understand:



If we take a look back at the CSS produced, Susy removed the final `margin-right` from `.sidebar` because we supplied it with a `last` keyword. This is what allows the final item to be placed on the same row on an `after` `gutter-position` setting.

You'll also notice that the final item is floated to the right instead of the left. This is done to address subpixel rounding errors on browsers. We will talk more about subpixel rounding errors in a later chapter.

A Quick Wrap Up

We learned about two frequently used mixins in this chapter. It's important that you understand what these two mixins do and how to use them before you move on to subsequent chapters. The two mixins are: `container()` and `span()`.

Susy Context

We briefly touched on `$context` while explaining the `span()` mixin in the previous chapter. Context is arguably the most important concept that you will need to understand when working with Susy.

You will learn:

- What context is
- How to work with context in nested layouts
- How to identify the context used

In later chapters, you'll also discover that context can be used with other Susy mixins to help you create your grid. Let's begin by understanding what context is.

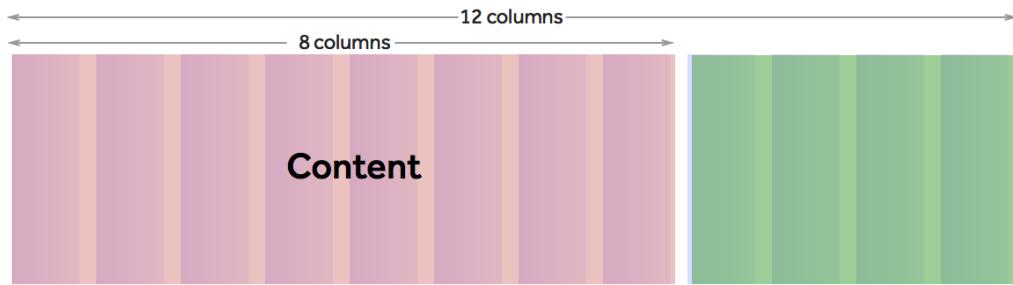
What is Context

Consider the following `span()` mixin.

```
.content {  
  @include span(8 of 12);  
}
```

We know from the previous chapter that `$context` is the argument that follows the `of` keyword. In this case, the context is 12.

This is what the code means when we translate it into a picture:



The simplest way to understand context is: **Context is the number of columns in the parent element.**

Using Context in Nested Layouts

Nested layouts are layouts where you have to create a Susy grid within another Susy grid. They are very common in real world situations.

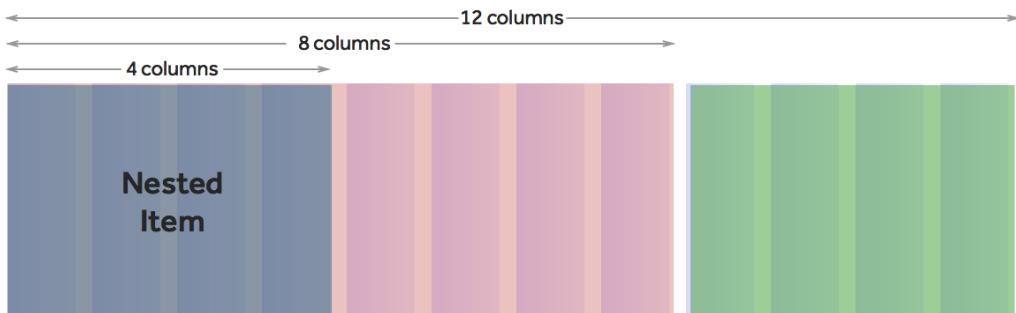
It can look something like this:

```
<div class="wrap">
  <div class="content">
    <!-- nested item -->
    <div class="inner-content">Nested Item</div>
  </div>
  <div class="sidebar"></div>
</div>
```

If `.inner-content` has to take up a specific number of columns within `.content`, you know you have a nested Susy grid item to deal with.

This is an area where most people run into problems when using Susy. They run into problems because they lack the understanding of how context in Susy works. Once you understand this concept, the rest of the book will be a breeze.

Say we want to translate the above HTML into this:



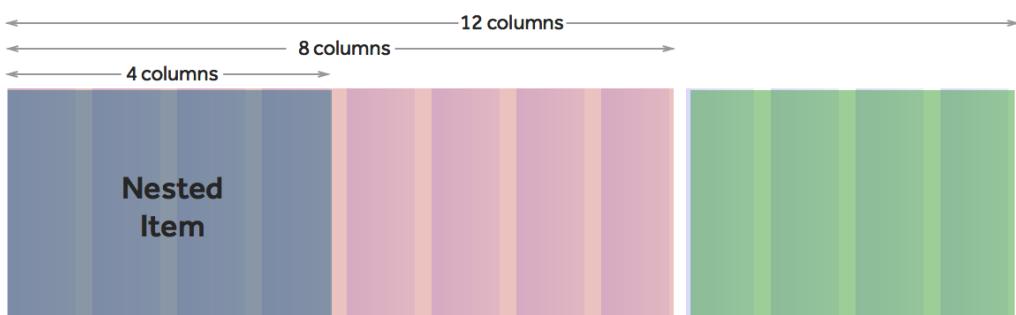
We know that `.wrap` is 12 columns, `.content` is 8 columns and `.inner-content` is 4 columns.

Since we know that context refers to the number of columns in the parent element, we know the code should be:

```
// .content takes up 8 columns. It's parent, .wrap, takes up 12
columns
.content {
  @include span(8 of 12);
}

// .inner-content takes up 4 columns. It's parent, .content,
takes up 8 columns

.inner-content {
  @include span(4 of 8);
}
```



It's easy to understand once you know the theory behind it. You just have to count the number of columns! :)

Context is incredibly important and it appears in every Susy mixin you use.

Most of the time though, context isn't as obvious as the above code. This is because once we get to a higher level with Susy, we want to stop repeating the context as much as possible and keep our Sass code DRY (Don't repeat yourself).

We have to learn where and how to use context for real world programming.

Identifying Context In Any Situation

We already know that every Susy mixin or function has a context baked into it. The most basic way to find context is to look for it within the `span()` mixin, or any other Susy mixin you use. You spot context by looking out for the number following the `of` keyword.

```
.content {  
  @include span(8 of 12);  
}
```

The context is 12 in this case.

As we move along, you will start to notice that we intentionally opt to not write the context for the reasons mentioned above. In those cases, you will see the `span()` mixin contain only the `$span` value, or the `last` keyword.

```
.content {  
  @include span(8);  
}  
  
.sidebar {  
  @include span(4 last);  
}
```

Susy will look upwards into your code for any trace of `$context` in these cases. The first place it will look is usage of a `nested()` or `with-layout()` mixin that wraps around the `span()` mixin.

Both `with-layout()` and `nested()` mixins are convenient mixins to help us switch out information passed into the `$susy` map when the `span()` mixin is called.

The `nested()` mixin is a subset of the `with-layout()` mixin that helps us change the context easily:

Here's an example:

```
@include nested(8) {
  .inner-content {
    @include span(4);
  }
}
```

Since we wrapped a `nested(8)` above the `span(4)`, the eventual output will mean the same as

```
.inner-content {
  @include span(4 of 8);
}
```

If you have to change other parts of the `$susy` map just for this part of the code, you can use the `with-layout()` mixin instead, and apply another `$susy` map to it. For example:

```
$new-susy-map: (
  columns: 16,
  gutters: 1/22,
  gutter-position: split,
);

@include with-layout($new-susy-map) {
  // altered grids within here
}
```

We will cover the `with-layout()` mixin in more detail in later chapters. Let's stick to `nested()` for now.

If no `with-layout()` or `nested()` mixin is found wrapping the `span()` mixin, Susy will automatically go all the way back up to the `$susy` map to get the context.

```
$susy: (
  columns: 12,
);

.inner-content {
  @include span(4);
}
```

The above code would create an output that is the same as this:

```
.inner-content {
  @include span(4 of 12);
}
```

We know that this is a wrong demonstration and `.inner-content` should be `span(4 of 8)`. Can you find the problem and fix it yourself?

A Quick Wrap Up

We explored what context really means when working with Susy. We also went through how to identify context and how it can be used to help create nested grids easily.

Once you get this concept cleared up, everything within Susy will become a breeze. Look forward to it :)

A More Complex Layout

We're building a layout that resembles what you may build in a real situation. Here's where the real fun begins.

There is a lot to cover in this chapter. We will split it into two parts to thoroughly explain how you could do the similar things in different ways with Susy.

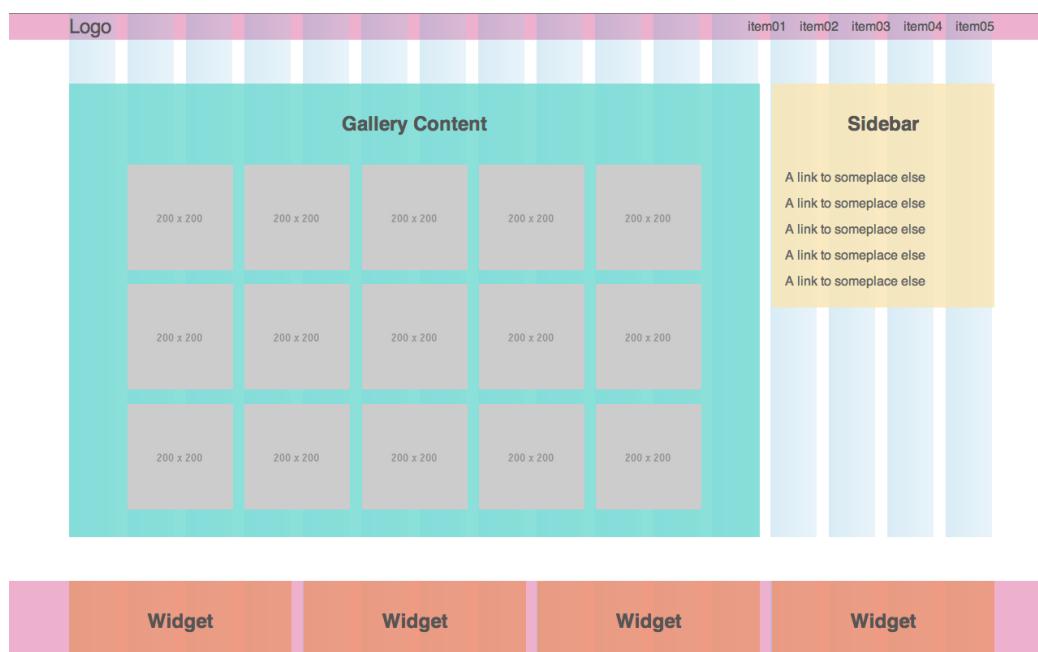
In this chapter, you will learn:

- When to use Susy, and when not to use Susy
- How to center a Susy grid item with the `span()` mixin
- How to center a Susy grid item with the `span()` and `gutter()` functions

In the next 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 in these two chapters:



Taking Care of CSS

We're speeding up the CSS again this chapter. There are some nuggets here that you might want to add into your own styles.

Here's what I did in addition to giving height and background colors to the layouts:

- removed margins and paddings from all ``, `` and `` elements
- removed list styles from all `` elements
- added `max-width: 100%` and `height: auto` to make images responsive.

```
// Scss
h2 {
  padding: 1rem 0;
  text-align: center;
  color: #555;
}

ul, ol {
  margin: 0;
  padding: 0;
}

li {
  list-style: none;
}
img {
  max-width: 100%;
  height: auto;
}

.site-header, .site-footer {
  background: rgba(234, 159, 195, 0.8);
}
```

```
.site-header {  
  a {  
    color: #555;  
    text-decoration: none;  
  }  
}  
  
.content {  
  margin-top: 5vh;  
  padding-bottom: 1rem;  
  background: rgba(113, 218, 210, 0.8);  
}  
  
.sidebar {  
  margin-top: 5vh;  
  background: rgba(250, 231, 179, 0.8);  
  padding-bottom: 1rem;  
  a {  
    color: #666;  
    padding-left: 1rem;  
    line-height: 2;  
    text-decoration: none;  
  }  
}  
  
.widget {  
  background: rgba(240, 150, 113, 0.8);  
}  
  
.site-footer {  
  margin-top: 5vh;  
}
```

The Susy Map

As with all Susy projects, you begin with the `$susy` map.

You can see from the layout that there are a total of 16 columns in this project. Everything else remains the same as with the previous layout.

```
// Scss
$susy: (
  columns: 16,
  container: 1140px,
  global-box-sizing: border-box,
  debug: (image: show)
);

@include border-box-sizing;

.wrap {
  @include container();
}
```

The Header Section

Let's break down the HTML as we go along, beginning with the header.



The header contains a logo and some navigational links. It has a background, which takes up 100% of the browser width. In order for this to happen, we have to keep the grid container (`.wrap`) within the `<header>` .

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

We can see that the logo is flushed to the left of the grid while the navigational links are flushed towards the right of the grid. This would mean that both the `.logo` and `<nav>` are wrapped within `.wrap` .

```
<!-- HTML -->
<header class="site-header">
  <div class="wrap">
    <div class="logo"><a href="#">Logo</a></div>
    <nav>
      <ul>
        <li><a href="#">item01</a></li>
        <li><a href="#">item02</a></li>
        <li><a href="#">item03</a></li>
        <li><a href="#">item04</a></li>
        <li><a href="#">item05</a></li>
      </ul>
    </nav>
  </div>
</header>
```

Since the logo is flushed left and the navigational links are flushed right, there is no need to use Susy to create their styles. We can simply apply

`float: left` to `.logo` and a `float: right` to `<nav>`.

```
.logo {
  float: left;
  line-height: 2rem;
  font-size: 1.5rem;
}

nav {
  float: right;

  li {
    list-style: none;
    float: left;
    margin-left: 1em;
    line-height: 2rem;
  }
}
```

[View Source Code](#)

As you can see, you don't have to use Susy with every element. You can always use standard CSS if Susy becomes overkill.

The Content and Sidebar

The `.content` and `.sidebar` sections are similar to what we had in the previous chapter. This time round, let's add some html elements into these sections.


```
</div>
```

Sass code for `.content` and `.sidebar` would be as follows since `.content` takes up 12 of 16 columns and `.sidebar` takes up 4 of 16 columns:

```
// Scss
.content {
  @include span(12 of 16);
}

.sidebar {
  @include span (4 of 16 last);
}
```



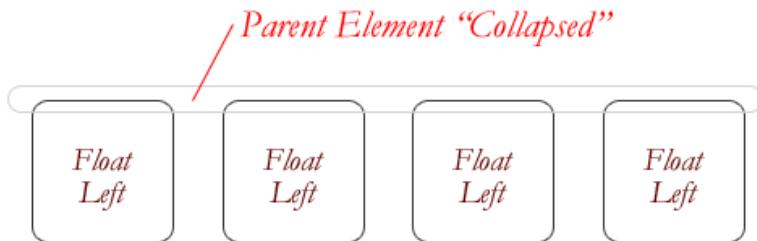
[View Source Code](#)

The Gallery Within Content

Susy layouts are predominantly made with floats. This brings me to the float collapse problem. It happens when every child item within the parent element is floated. Chris Coyier explains this problem the best:

“One of the more bewildering things about working with floats is how they can affect the element that contains them (their “parent”

element). If this parent element contained nothing but floated elements, the height of it would literally collapse to nothing. This isn't always obvious if the parent doesn't contain any visually noticeable background, but it is important to be aware of”.



“Collapsing almost always needs to be dealt with to prevent strange layout and cross-browser problems. We fix this problem by clearing the float after the floated elements in the container but before the close of the container”.

[All About Floats](#), Chris Coyier

The parent element’s height will collapse to nothing all of its child elements are floated. We have to fix this float collapse problem with a `clearfix` mixin.

This `clearfix` mixin will include a pseudo element that will clear the floated contents within the container right before the container closes.

Place this mixin at the beginning of your Sass file:

```
// Scss. Clearfix Mixin
@mixin cf {
  &:after {
    content: " ";
    display: block;
    clear: both;
  }
}
```

We know that `.gallery` is the container for `.gallery__items` and that we are going to use the `span()` mixin on each `.gallery__item`. This means that all child elements within `.gallery` are floated. We will need to give `.gallery` a clearfix.

```
// Scss
.gallery {
  @include cf;
}
```

Before working on the rest of `.gallery`, let's take a look at the final layout again:



From the image, you can see that the `gallery__items` fit onto a 10 column width. This would mean that we need to position `.gallery` in such a way that it is centered within `.content`, and that its width takes up 10 columns exactly.

There are two ways to center `.gallery` in the middle of the 12-column `.content`. We will walk through all both ways. Before that do that, we need to learn more about other Susy functions that Susy provides us with.

These functions can help us tremendously in achieving what we want. They are the `span()` function and the `gutter()` function.

The `span()` Function

You may be thinking, isn't `span` a mixin?

That is correct. But Susy also provides a function that uses the same name. The `span` function takes in exactly the same arguments as the `span` mixin, but it returns only the value of `width` instead of writing 3 properties.

Here's a comparison between the `span` function and the `span` mixin in use:

```
// SCSS
.span-mixin {
  @include span(12 of 16);
}

.span-function {
  width: span(12 of 16);
}
```

```
/* CSS */
.span-mixin {
  width: 74.68354%;
  float: left;
  margin-right: 1.26582%;
}

.span-function {
  width: 74.68354%;
}
```

Note: If you're unsure of how to work with functions and mixins, turn back to [chapter 3](#), it's explained there :)

The `span` function returns the width of the desired `$span` and can be used in any property. Here's an example where the function is used on a `margin` property:

```
// SCSS
.span-function {
  margin-right: span(1);
}
```

```
/* CSS */
.span-function {
  margin-right: 5.06329%;
}
```

Here, we've created a `margin-right` property and given it a value equal to one column.

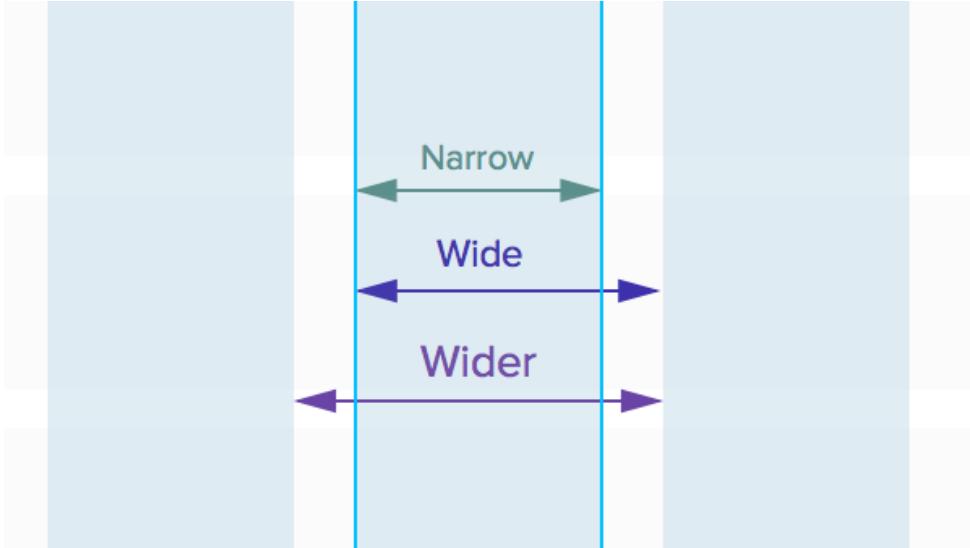
There is one more useful argument that you can give to both the `span()` function and the `span()` mixin: `$spread`.

The `$spread` Argument

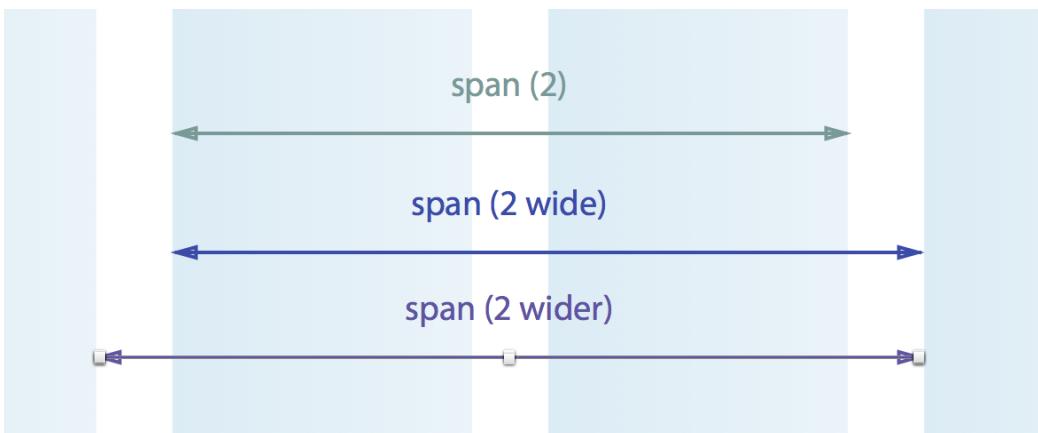
`$spread` is an optional argument that allows you to change the `width` output in both the `span()` function and mixin to explicitly state whether the `width` should be expanded to include one or two more gutters.

`$spread` has 3 different options for you to choose from.

- Narrow (*default*)
- Wide
- Wider



When calculating the width of a `span`, Susy will include all internal gutters by default. `wide` and `wider` simply adds one or two more gutters into the width.



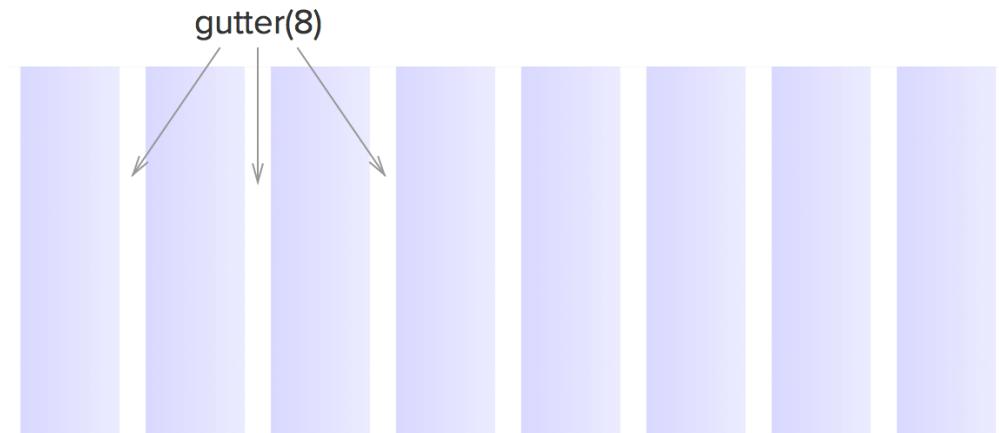
The `narrow` option works best most of the time. You'll only use `wide` or `wider` if you need to add a gutter or two to the `width` calculations.

The `gutter()` Function

The `gutter()` function, as its name suggests, is a function to output the width of a gutter. It takes one argument: `$context`.

```
// Scss
.test {
  width: gutter($context);
}
```

This `$context` allows the `gutter()` function to calculate and output the width of one gutter. In a container with 8 columns, one gutter size would be `gutter(8)`.



If you didn't give the `gutter` function a context, it will look for the context in other areas as explained in the previous chapter.

Let's see how we can use these two functions to center the layout now.

Centering the Gallery

There are two ways to center `.gallery`.

1. Setting the width of `.gallery`
2. Adding padding to `.gallery`

Both methods work perfectly fine. Feel free to use the two of them interchangeably.

Since we're working on centering the gallery for the first time, let's give `.gallery` a temporary background color of `rgb(200, 200, 2)` to detect its exact size and position.

```
// Scss
.gallery {
  @include cf;
  background: rgb(200, 200, 2);
}
```



Method 1: Setting the Width of Gallery

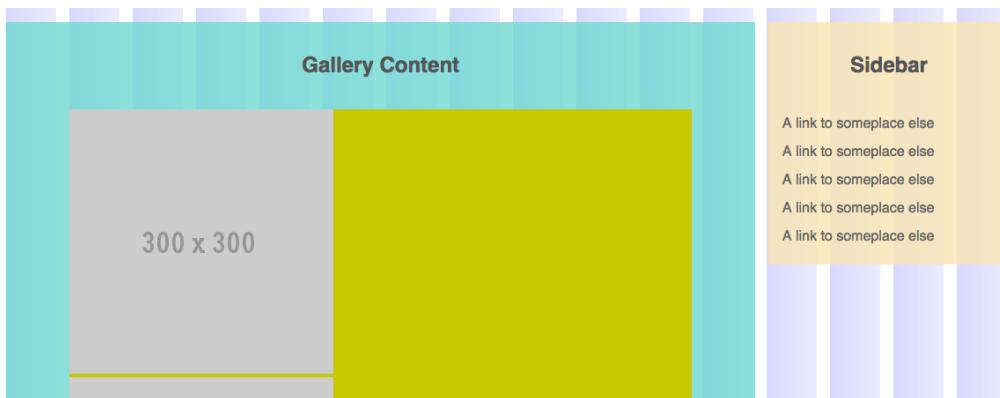
The first method is to set the width of the parent container to 10 columns. We can do this with either the `span()` mixin or `span` function. Let's use the `span()` function to keep the code DRY since we only require the `width` property.

```
// Scss
.gallery {
  width: span(10 of 12);
}
```



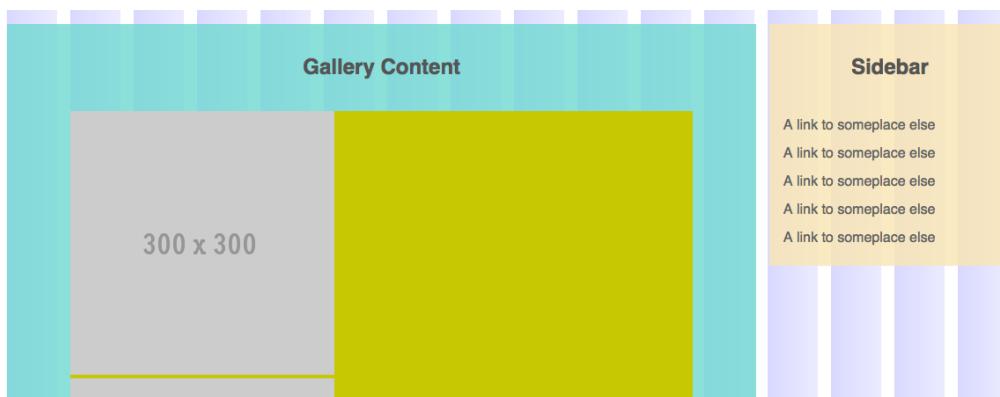
We can center the container the same way as we centered the `.wrap` container by setting `margin-left` and `margin-right` to auto.

```
// Scss
.gallery {
  width: span(10 of 12);
  margin-left: auto;
  margin-right: auto;
}
```



Alternatively, since we know the context, we can push the `.gallery` from one side of the container with the `span` function, or with a combination of the `span()` and `gutter()` functions.

```
// Scss
.gallery {
  width: span(10 of 12);
  margin-left: span(1 wide of 12);
  // OR
  // margin-left: span(1 of 12) + gutter(12);
}
```



[View Source Code](#)

Method 2: Adding Padding to Gallery

The second method is to add `padding-left` and `padding-right`, that is the width of one column plus one gutter, to `.gallery`. This method only works if you are using the `border-box` box-sizing property, which we are if you've been following along.

If we choose to add padding to the gallery, we have to add a background to `.gallery__item` instead to visualize how much space `.gallery__item` has at 100% width.

```
// Scss
.gallery__item {
  background: rgb(100, 100, 200);
}
```

Again, we can either use a combination of the `span()` and `gutter()` functions or we can use the `span()` function with the `$spread` keyword.

```
// Scss
.gallery {
  padding: 0 span(1 wide of 12);
  // OR
  // padding: 0 span(1 of 12) + gutter(12);
}
```



[View Source code](#)

A Quick Wrap Up

We have gone through the basics of creating a more complex layout in this chapter. We have also learned how to center any HTML element within Susy. You may also have discovered that there are multiple ways of achieving the same outcome when using Susy, and that makes Susy flexible enough to adapt to your unique requirements.

We will move on to complete the layout in the next chapter and you will learn more about how to create galleries with Susy.

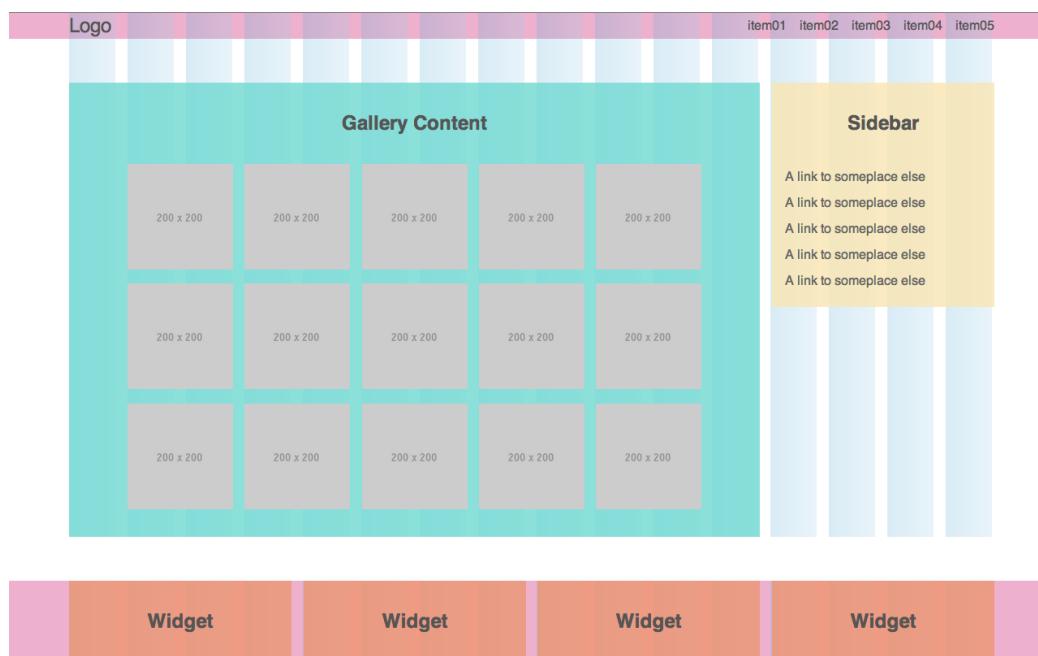
A More Complex Layout (Part 2)

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:

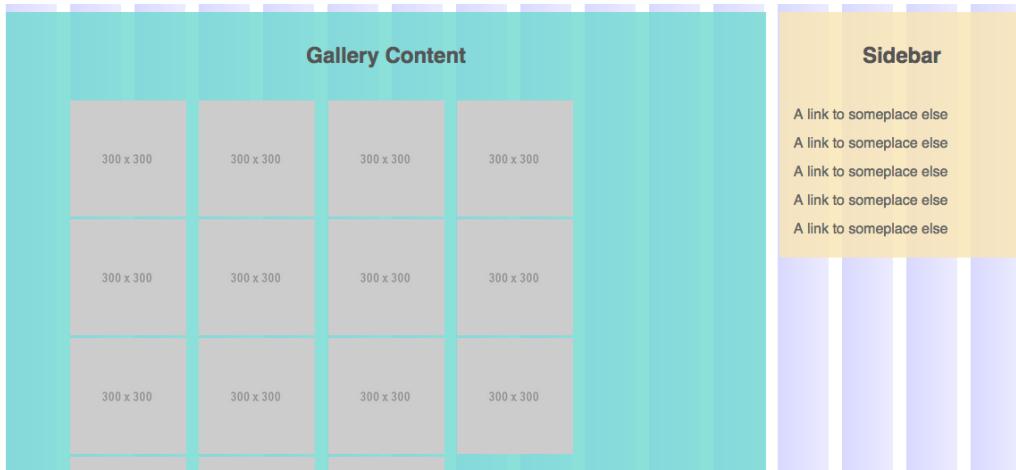


Building The Gallery Content

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.

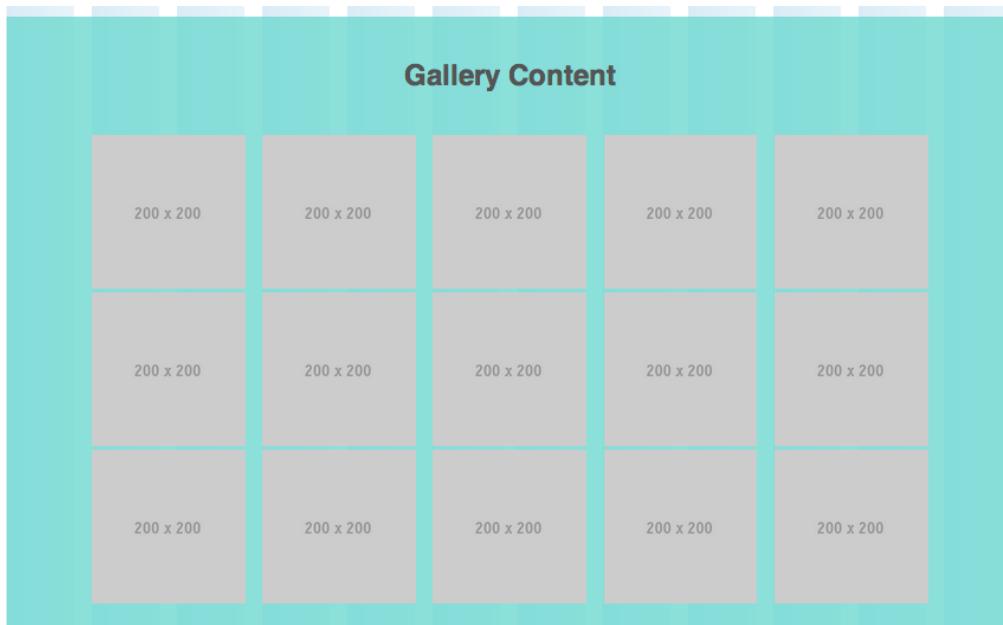
```
// Scss
.gallery__item {
  @include span( 2 of 10 );
}
```



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

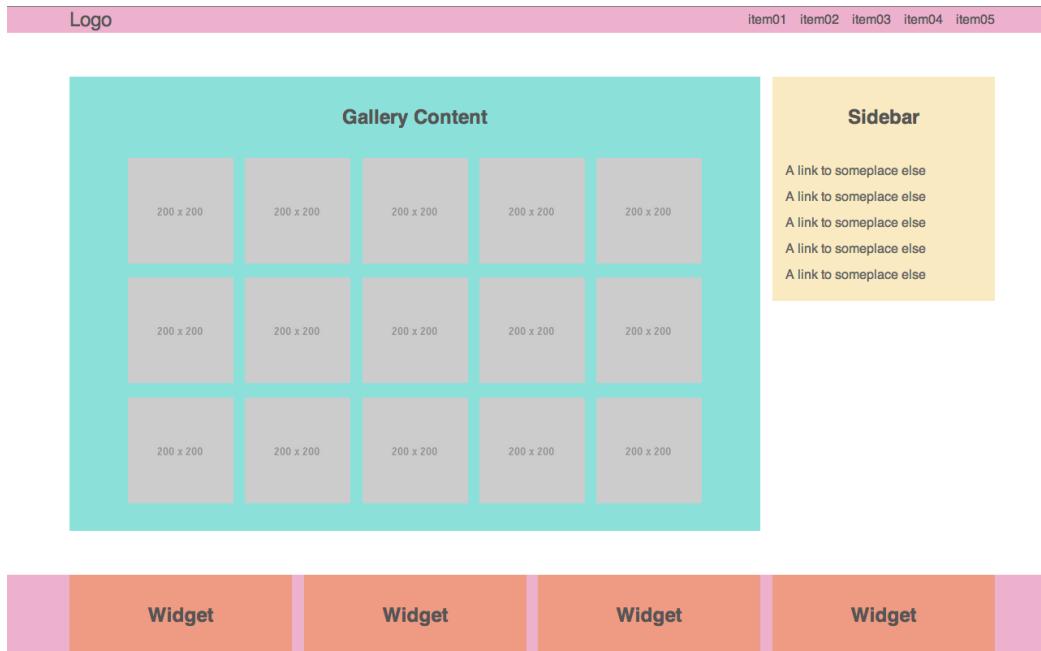
The `gallery()` mixin is specially created to make galleries with Susy. It takes in the same arguments as the `span()` mixin and can only create gallery items that take up the same number of columns.

```
// 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 */
.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.

Media Queries

Knowing how to develop responsive websites has become an indispensable skill for today's web developer.

What makes a website responsive then? How should we code? These are important questions that we will cover in this chapter.

You'll learn:

- What is required to make responsive websites
- How to code responsive websites, mobile-first

Things Required For a Responsive Website

There are only two things you will need to make a website responsive.

1. Meta Viewport Tag
2. Media Queries

Meta Viewport Tag

Hand-held devices today usually have a screen width that is larger than the actual width of the device. iPhone 5s for example, has a device resolution of 640px x 1136px. The actual screen size however, is half of that, at 320px x 568px.

Browsers render the width according to screen resolution by default. They will think our iPhone 5s has a width of 640px instead of 320px. This would mean that everything will be zoomed out to be half its intended size on an iPhone 5s if we don't do something about it.

To combat this, we need to add a meta viewport tag to the `<head>` of the website. This will tell the browser to render the width of the page as the width of its own screen.

```
<head>
  <meta name="viewport" content="width=device-width, initial-
  scale=1">
</head>
```

Now, the iPhone 5s will render at 320px where we intend it to.

Media Queries

If you resize your browser on a responsive website, you will notice that the layout of the website changes at certain points. These are called breakpoints.

Media queries are used to specify where these breakpoints are, and how the browser should respond when the conditions are matched. They have the following syntax:

```
/* CSS */
@media [<media-type> and] (<condition>) {
  /* properties here */
}
```

`@media` is the keyword for media queries. It signifies the start of a media query.

`<media-type>` is an optional argument that tells browsers to filter for a specific media type. The most common media types are `all`, `screen` and `print`. The `print` media type only applies when the page is printed out while `screen` works when the site is viewed on the screen. `<media-type>` defaults to `all` when not specified, will apply to both `print` and `screen`.

`condition` is a required argument in a media query that allows us to specify how the browser should render a page if the condition is true. They are written in a property and value pair like:

```
@media (min-width: 600px) {  
    /* This applies when the browser width is equal to or larger  
    than 600px */  
}
```

You can chain more than one conditions with the `and` keyword, like:

```
@media (min-width:600px) and (max-width: 1200px) {  
    /* This applies only when the browser width is BETWEEN 600px  
    and 1200px */  
}
```

Note: There are more conditions that you can potentially use with media queries. We will only cover `min-width` and `max-width` since they will be used the most.

Media queries have to be written as the first level wrapper in a CSS file. As such, every media query declaration would have the following format:

```
/* CSS */  
@media (min-width:600px) {  
    .content {  
        /* Content properties */  
    }  
}
```

Writing this way could become problematic when we have hundreds of selectors that use a specific media query. People used to throw in these media queries together at the end of the css file:

```
/* Standard CSS properties here */

@media (min-width: 600px) {
    .selector1 {}
    .selector2 {}
    .selector3 {} /* ... */
}

@media (min-width: 1200px) {
    .selector1 {}
    .selector2 {}
    .selector3 {} /* ... */
}
```

This way of coding decouples the media queries from the selectors themselves, and makes it hard for us to understand the code when we look at it at a later time.

There are only 3 selectors above, imagine if you have over 200 selectors in each media... How much of a nightmare would that be?

Thankfully, as of Sass v3.2, media queries can be placed within the selectors themselves:

```
// Scss
.content {
    @media (min-width:600px) {
        @include span(3 of 4);
    }

    @media (min-width:1200px) {
        @include span(8 of 12);
    }
}
```

This way of coding couples the styles from the same selector at different breakpoints together, allowing you to easily understand what is going on.

Sass automatically translates it into the CSS code format when compiled:

```
/* CSS */
@media (min-width: 600px) {
  .content {
    /* CSS properties at 600px */
  }
}
@media (min-width: 1200px) {
  .content {
    /* CSS properties at 1200px */
  }
}
```

You may be concerned about performance since we're introducing multiple media queries when we code this way. You don't have you worry.

This is because the difference in rendering time between a stylesheet with 40 media queries and another with 2000 media queries is only a mere 100 milliseconds difference in a [test](#) done by [Aaron Jenson](#).

Since we now know how to work with media queries, let's find out how to write them, mobile first.

Mobile First Media Queries

Mobile-first and desktop-first media queries have subtle differences.

A mobile-first approach to styling means that styles are applied first to mobile devices. Advanced styles and other overrides for larger screens are then added into the stylesheet via media queries.

This approach uses `min-width` media queries.

Here's a quick example:

```
// This applies from 0px to 600px
body {
    background: red;
}

// This applies from 600px onwards
@media (min-width: 600px) {
    body {
        background: green;
    }
}
```

In the example above, `<body>` will have a red background below 600px. Its background changes to green at 600px and beyond.

On the flipside, a desktop-first approach to styling means that styles are applied first to desktop devices. Advanced styles and overrides for smaller screens are then added into the stylesheet via media queries.

This approach uses `max-width` media queries.

Here's a quick example:

```
// This applies from 600px onwards
body {
    background: green;
}

// This applies from 0px to 600px
@media (max-width: 600px) {
    body {
        background: red;
    }
}
```

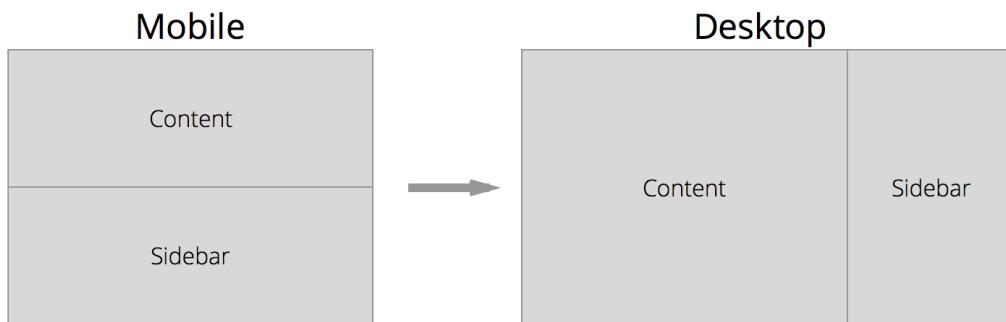
`<body>` will have a background colour of green for all widths. If the screen goes below 600px, the background colour becomes red instead.

Why Code Mobile-first?

Code for larger screens is usually more complicated than the codes for smaller screens. This is why coding mobile first helps simplify your code.

Consider a situation where you have a content-sidebar layout for a website.

.content takes up a 100% width on mobile, and 66% on the desktop.



Most of the time, we can rely on default properties to style content for smaller screens. In this case, a <div> has a width of 100% by default.

If we work with the mobile-first approach, the Sass code will be:

```
.content {  
    // Properties for smaller screens.  
    // Nothing is required here because we can use the default  
    styles  
  
    // Properties for larger screens  
    @media (min-width: 800px) {  
        float: left;  
        width: 60%;  
    }  
}
```

If we go with the desktop-first approach instead, we will have to restore the default properties for smaller viewports most of the time. The Sass code for the same result is:

```
.content {  
    // Properties for larger screens.  
    float: left;  
    width: 60%;  
  
    // Properties for smaller screens.  
    // Note that we have to write two default properties to make  
    the layout work  
    @media (max-width: 800px) {  
        float: none;  
        width: 100%;  
    }  
}
```

With this one example, we save two lines of code and a few seconds of mind-bending CSS. Imagine how much time and effort this will save you if you worked on a larger site.

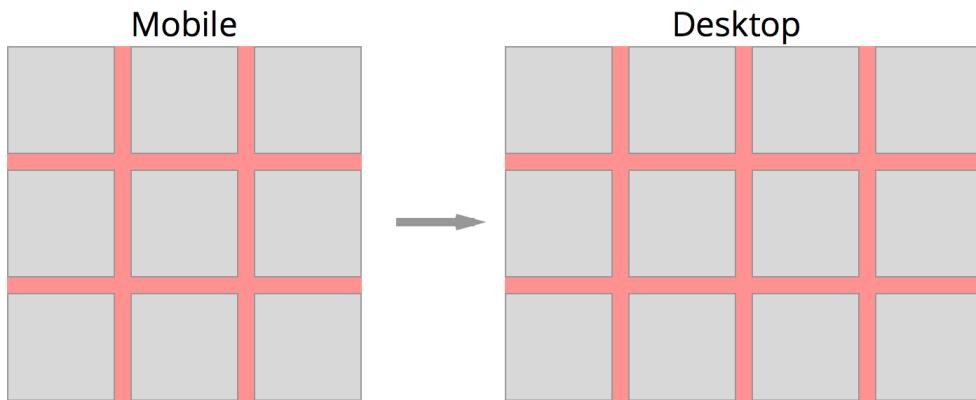
Most of the time `min-width` queries would be enough to help you code a website. There are however instances where a combination of both `min-width` and `max-width` queries helps to reduce complications that pure `min-width` queries cannot hope to achieve.

Let's explore some of these instances.

Using Max-width Queries With A Mobile-First Approach

`Max-width` queries come into play when you want styles to be constrained below a certain viewport size. A combination of both `min-width` and `max-width` media queries will help to constrain the styles between two different viewport sizes.

Consider a case of a gallery of thumbnails. This gallery has 3 thumbnails in a row on a smaller viewport and 4 items in a row on a larger viewport.



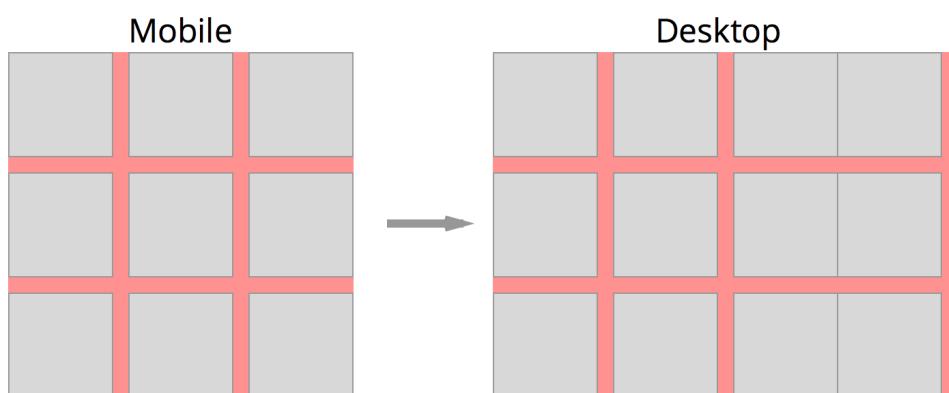
```
.gallery__item {
  @include span(4 of 12);
  margin-bottom: gutter(12);
}
```

We will also have to give the final (3rd item) on the row the `last` keyword to make sure it doesn't get pushed down into the next column.

```
.gallery__item {
  @include span(4 of 12);
  margin-bottom: gutter(12);
  &:nth-child(3n) {
    // `last()` is a mixin that removes the right margin
    @include last;
  }
}
```

This code must also work for the case where there are four items in the row. If we go according to the min-width query we had above...

```
.gallery__item {  
  @include span(4 of 12);  
  margin-bottom: gutter(12);  
  &:nth-child(3n) {  
    @include last;  
  }  
  
  @media (min-width: 800px) {  
    @include span(3 of 12);  
    &:nth-child (4n) {  
      @include last;  
    }  
  }  
}
```



This doesn't work properly because we specified that every 3rd item should have a `margin-right` of 0px. This property gets cascaded towards a larger viewport and breaks the pattern we wanted.

We can fix it by resetting the `margin-right` property of every 3rd item to the correct gutter:

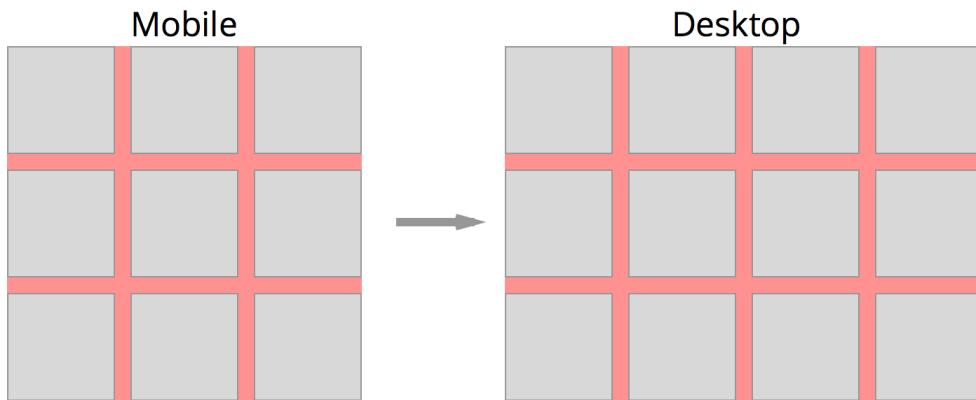
```
.gallery__item {
  // ...
  @media (min-width: 800px) {
    // ...
    &:nth-child(3n) {
      margin-right: gutter(12);
    }
    &:nth-child(4n) {
      margin-right: 0%;
    }
  }
}
```

It's not a very nice approach since we are repeatedly redeclaring and removing the `margin-right` property of several items with the `last()` mixin.

A better way is to constrain `nth-child(3n)` selector within its rightful viewport by using a `max-width` query.

```
.gallery__item {
  margin-bottom: gutter(12);
  @media (max-width: 800px) {
    @include span(4 of 12);
    &:nth-child(3n) {
      @include last;
    }
  }

  @media (min-width: 800px) {
    @include span(3 of 12);
    &:nth-child(4n) {
      @include last;
    }
  }
}
```



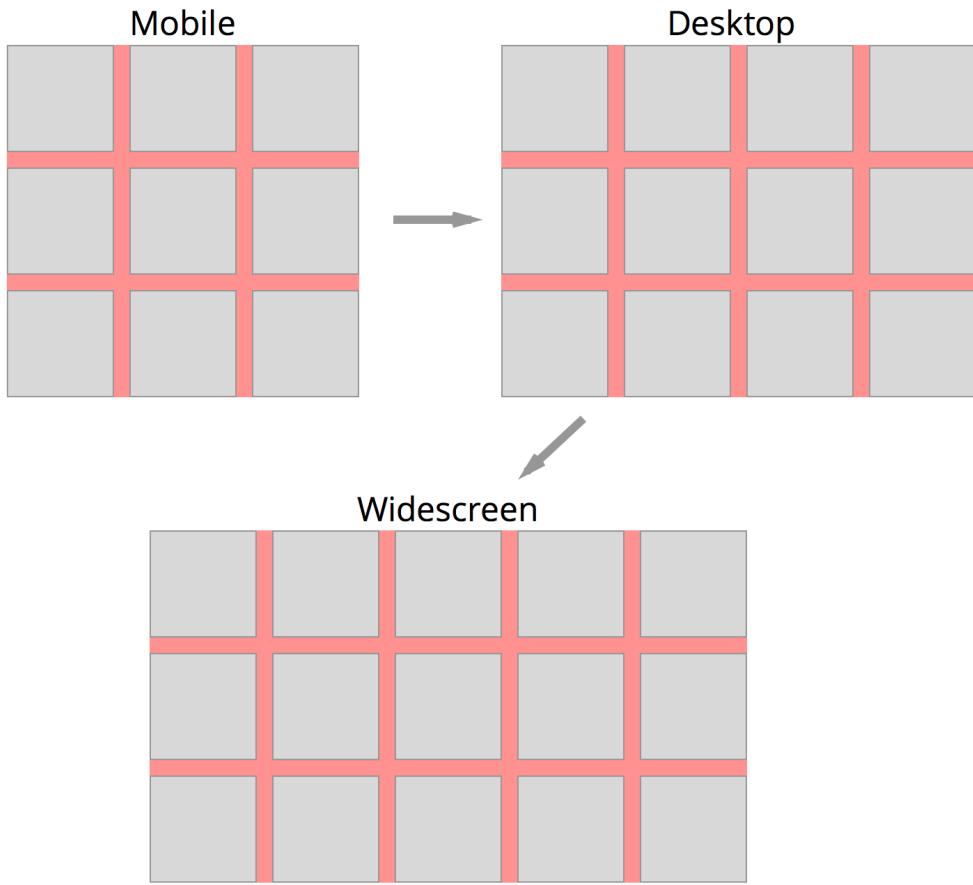
This works because the `max-width` property limits the selectors to below 800px and the styles given within will not affect styles for any other viewports.

Now imagine if you have a larger viewport and you wanted to show 5 items per row in the gallery. This is when a combination of `min` and `max-width` queries come together.

```
.gallery__item {
  margin-bottom: gutter(12);
  @media (max-width: 800px) {
    @include span(4 of 12);
    &:nth-child(3n) {
      @include last;
    }
  }

  // Combining both min-width and max-width queries
  @media (min-width: 800px) and (max-width: 1200px){
    @include span(3 of 12);
    &:nth-child (4n) {
      @include last;
    }
  }

  @media (min-width: 1200px){
    @include span(3 of 15);
    &:nth-child(5n) {
      @include last;
    }
  }
}
```



That in the nutshell, is how to write mobile-first CSS.

A Quick Wrap Up

`Min-width` media queries are extremely helpful when it comes to coding responsive websites because it reduces code complexity. `Min-width` queries are, however, not the solution to every problem as you can see from the examples above. It can sometimes be beneficial to add `max-width` queries to help keep things cleaner.

Breakpoints with Susy

Writing media queries with `@media` can become tiring quickly because of the amount of text you have to type each time. Luckily, Susy has a way to help you write shorter media queries with the correct context, the `susy-breakpoint()` mixin. The `susy-breakpoint()` requires the use of the [Breakpoint library](#).

We will be using the `susy-breakpoint()` mixin a lot in future chapters, so it will be good to thoroughly understand what it does and how to use it before we move onto creating the responsive website.

Note: As of Susy v2.1.3, the `susy-breakpoint()` mixin no longer requires the Breakpoint-sass library. However, since breakpoint-sass is still very useful for responsive layouts, we will still work through how to use it.

In this chapter, you'll learn:

- How to use the Breakpoint library
- How to use the `susy-breakpoint()` mixin to create responsive websites

Installing The Breakpoint Library

There are three ways to install Breakpoint.

1. Install the Breakpoint Gem (This method requires you to use Compass for your project)
2. Download the Breakpoint repository
3. Use Bower

Method 1 – Install the Breakpoint Gem

The first method is to install the Breakpoint gem through the terminal. This is exactly the same as how you would have installed Sass or Susy.

```
# command line  
$ sudo gem install breakpoint
```

You'll then have to configure `config.rb` to require breakpoint.

```
# config.rb  
require 'breakpoint'
```

Lastly, you will need to import Breakpoint into your Sass project.

```
// SCSS  
@import "breakpoint";
```

Method 2 – Download the Breakpoint Repo

The second method is to download the Breakpoint repository into your Sass folder and import it into your stylesheet. The actual file we want to import is called `_breakpoint.scss`, and this file is located within `breakpoint/stylesheets`.

Once you have located the file, you just have to import this `breakpoint.scss` into your Sass file.

```
// SCSS  
@import "path-to-breakpoint/stylesheets/breakpoint";
```

Method 3: Use Bower

Bower is a frontend dependency manager and is my favorite of all 3 methods. It allows you to manage frontend dependencies easily without complexities that

come with the other methods. You need to have Node.js and Node Package Manager installed onto your system if you want to use Bower.

You can install Breakpoint using Bower like this:

```
# Command line  
$ bower install breakpoint-sass --save-dev
```

Then, import Breakpoint in your stylesheets.

```
// SCSS  
@import "path-to-bower-components/breakpoint-  
sass/stylesheets/breakpoint";
```

Using Breakpoint

Breakpoint provides us with a `breakpoint()` mixin. The syntax for the mixin is as follows:

```
// SCSS  
@include breakpoint($arg){ @content };
```

`@content` refers to CSS properties that you use within the media query.

`$args` are arguments you give to Breakpoint to determine the media query that Breakpoint will produce. Breakpoint outputs different CSS depending on the number of arguments and the values of each argument given to it.

We'll cover the common media query usages that were mentioned in the previous chapter and how you could use `breakpoint` to achieve them.

Min-width Query

You have to provide breakpoint with one argument to create a `min-width` media query. This argument can be in pixels, ems or any other valid CSS unit.

```
// SCSS
.red-above-600px{
  @include breakpoint(600px) {
    color:red;
  }
}
```

Breakpoint will create the `min-width` media query for you.

```
/* CSS */
@media (min-width: 600px) {
  .red-above-600px {
    color: red;
  }
}
```

Min-width and Max-width Query

This isn't complicated either. You just have to give Breakpoint two widths as arguments. The first width given will be used for `min-width` condition and the second width given will be used for `max-width` condition.

```
// SCSS
.red-between-600px-and-900px {
  @include breakpoint (600px 900px) {
    color: red
  }
}
```

Breakpoint will convert these arguments into a `min-width` and a `max-width` query.

```
/* CSS */
@media (min-width: 600px) and (max-width: 900px) {
  .red-between-600px-and-900px {
    color: red;
  }
}
```

Max-width Query

Once in a while you'll want to use the `max-width` query. You do this by providing 2 arguments to breakpoint, a `max-width` string followed by the breakpoint.

```
.red-below-900px{
  @include breakpoint(max-width 900px) {
    color:red;
  }
}
```

Breakpoint will then output a `max-width` query.

```
/* CSS */
@media (max-width: 900px) {
  .red-below-900px {
    color: red;
  }
}
```

The `susy-breakpoint()` Mixin

`susy-breakpoint()` is a mixin that brings the Breakpoint library together with the Susy context. It has the following syntax:

```
// Scss
@include susy-breakpoint($breakpoint-args, $context) {
  @content
}
```

\$breakpoint-args are the arguments that you would give to the breakpoint() mixin we discussed above.

\$context is the same old Susy Context that we have been talking about since Chapter 6.

The susy-breakpoint() mixin is a convenience mixin that produces the following output:

```
// Scss
@include breakpoint($breakpoint-args) {
  @include nested($context) {
    @content
  }
}
```

Say you have an element that takes up 4 of 8 columns on a smaller viewport and 4 of 16 columns on a larger one. Its Sass code will be:

```
$susy: (
  columns: 8
);

.element {
  @include span(4);

  @include breakpoint(1200px) {
    @include nested(16) {
      @include span(4);
    }
  }
  // Alternatively, if you used the susy-breakpoint mixin:
  // @include susy-breakpoint(1200px, 16) {
  //   @include span(4);
  // }
}
```

A Quick Wrap Up

The `breakpoint()` mixin allows us to write media queries easily.

`susy-breakpoint()` brings it up a notch by adding a convenient argument that allows us to provide a `$context`, and it automatically uses the `nested()` mixin for us.

Both of these mixins help us keep our code DRYer and cleaner, but when should you use one or the other?

The answer is simple.

Use `susy-breakpoint()` when you need to write a media query and provide `$context` at the same time.

Use `breakpoint()` if you don't need to provide `$context`.

Simple ain't it? :)

There is just one more thing to understand before creating a responsive website with Susy. In the next chapter, you will learn everything you need to know about the Susy background grid, including how to change the number of columns at different breakpoints.

Susy Background Grid

The Susy background grid isn't used actually in the website. However, it's infinitely helpful to know how to work with it so you know that you're on the right track when building Susy layouts.

So far, we have only covered how to show the Susy background grid with the `container()` mixin, and you might have some questions about the background grid.

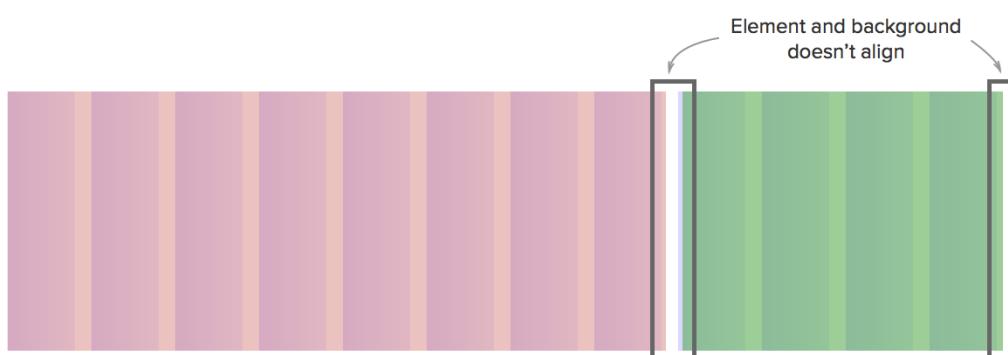
We're will go through everything you need to know about the background grid in this chapter.

You'll learn:

- Why the Susy background grid doesn't align properly with the elements on some browsers
- How to change the background grid at different breakpoints
- How to toggle the Susy background grid

The Background Grid Doesn't Align Perfectly

You may have noticed that sometimes the Susy grid background doesn't align perfectly to the elements produced by the Susy code. When that happens, you might start questioning if you are using Susy correctly.



Relax, you're not doing it wrong.

This happens because we have created subpixels with the CSS and [every browser deals with subpixels differently](#).

Subpixels are pixel values that are less than 1px, like 0.8px. When we create properties with percentages, we can't make sure that the elements are always going to be pixel perfect.

Unfortunately, we have to work with percentages to make responsive websites. This means our eventual CSS are littered with subpixels when rendered on browsers.

The Susy grid background will show up perfectly fine if you use Firefox and IE8+ to view the grid. This is because these browsers handle subpixels very well and output subpixels exactly as they are.

Webkit (Chrome and Safari) and Opera browsers, however, round down subpixels. A width value of 57.8px will be rounded down to 57px in these browsers and this causes a leftwards shift of all elements by 0.8px. Subpixel rounding errors like this are compounded when the number of items increase.

The Susy background grid is created as a background image on the `:before` element. Unfortunately, background images are subjected to subpixel rounding errors by the browsers. The larger the number of columns you use, the larger the rounding error compounded.

Susy tries to combat browser rounding errors naturally by floating the last item of each element to the right. This will ensure the right edge of your website is completely aligned.

However, this method only works if you have a small number of items per row. It would be more appropriate to use the Isolate Technique if you had more items, which we will cover in a later chapter.

For now, let's keep our attention on the background grid. Since we are going to work with responsive websites in the next chapter, we have to know how to change the number of columns in the grid background.

Changing The Background Grid at Different Breakpoints

You can change the number of columns in the grid background with the `show-grid()` mixin. It takes in one argument that you should be familiar with:

```
// Scss
@include show-grid($context);
```

This mixin can be used with different breakpoints to change how the background image is displayed. It is used in the following manner:

```
// Scss
$susy :(
  columns: 8,
  container: 1140px,
  debug: (image: show),
  global-box-sizing: border-box
);

.wrap {
  @include container();

  @include breakpoint(768px) {
    @include show-grid(9);
  }

  @include breakpoint(1024px) {
    @include show-grid(16);
  }
}
```

This code will produce a `.wrap` container that shows 8 columns on a screen below 768px, 9 columns on a screen between 768px to 1024px and 16 columns on a screen above 1024px.

Try resizing your browser now and you should see the grids changing at the specified breakpoints. :)

Now, let's talk about the rest of the properties you can change for the background grid.

Toggling The Background Grid Properties

Remember when we wrote the `$susy` map in the earlier chapters, we had to write the `debug` key a little differently to show the background grid?

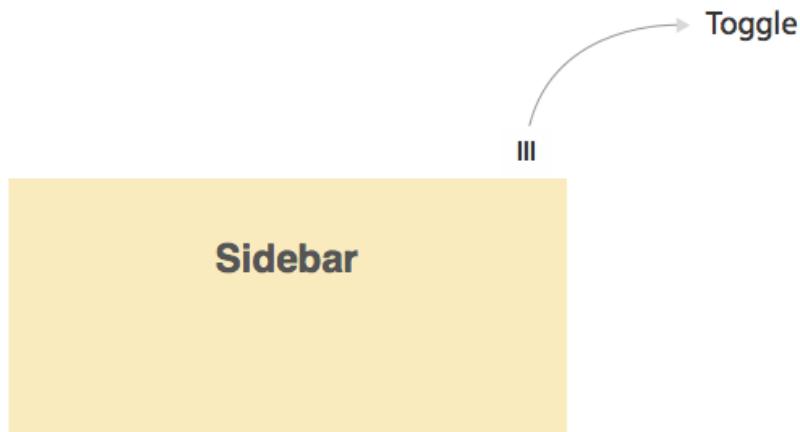
This is because Susy has other keys within `debug` that you could use to change how the background grid is displayed.

```
// Scss
$susy :(
  debug: (
    image: hide, // (hide | show | show-columns | show-baseline)
    color: rgba(#66f, .25),
    output: background, // (background | overlay)
    toggle: right top, // (left | right and top | bottom)
  )
);
```

The `image` key tells Susy whether to `hide` or `show` the image. `show` refers to showing both columns and the baseline grid (The baseline grid will only be shown if you are using Compass Vertical Rhythms).

`color` is the background colour of the columns. This can be changed into any other CSS color.

`output` allows you to change the background grid into an overlay that can be toggled on or off. When set to `overlay`, the debug grid will only show when you hover over the toggle.



`toggle` allows you to change the location of that toggle based on the options listed above.

This setting is mainly used for development and it allows you to customize how you would like the grid to appear. I recommend leaving the default setting.

A Quick Wrap Up

We have discussed everything we need to know about the Susy grid background in this chapter. It should have cleared up a lot of your doubts about the grid background and armed you with the knowledge to move onto building responsive layouts.

We will tackle a responsive layout in the next chapter.

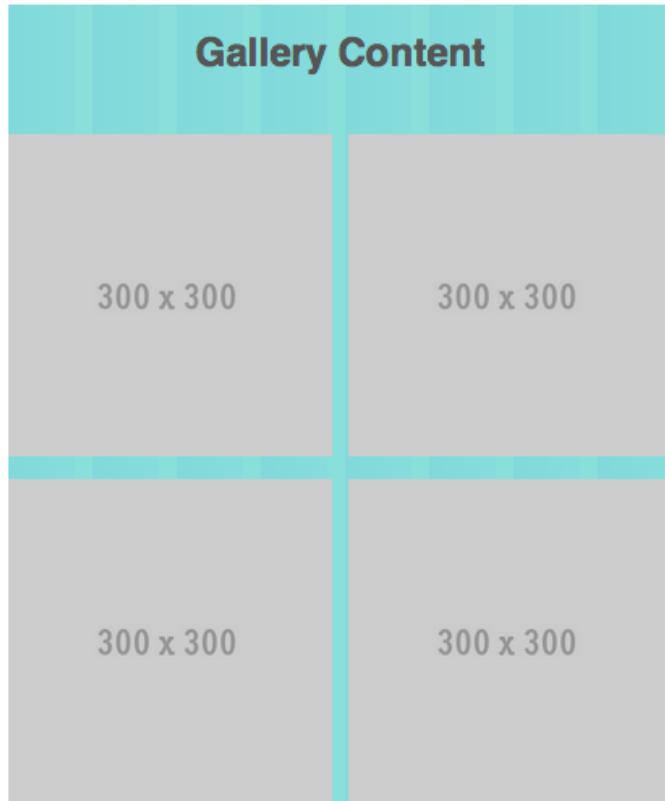
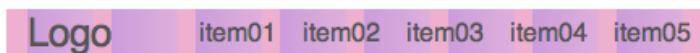
Building A Responsive Layout

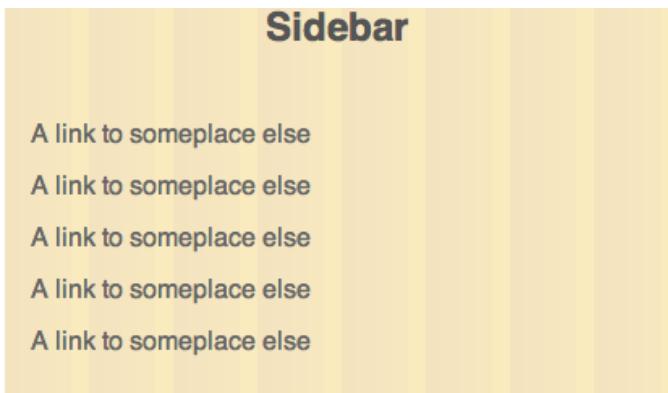
The objective of this chapter is to turn the complex layout we built in chapter 8 into a responsive website! The last 3 chapters have set the foundation for us to be able to do so.

You'll learn to build a responsive website with Susy and implement everything we have learned in the last 3 chapters.

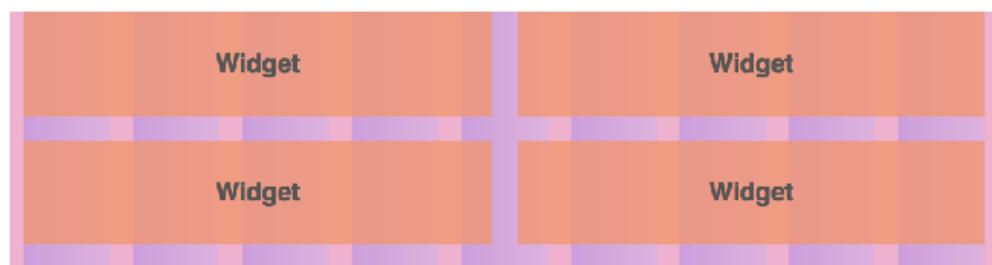
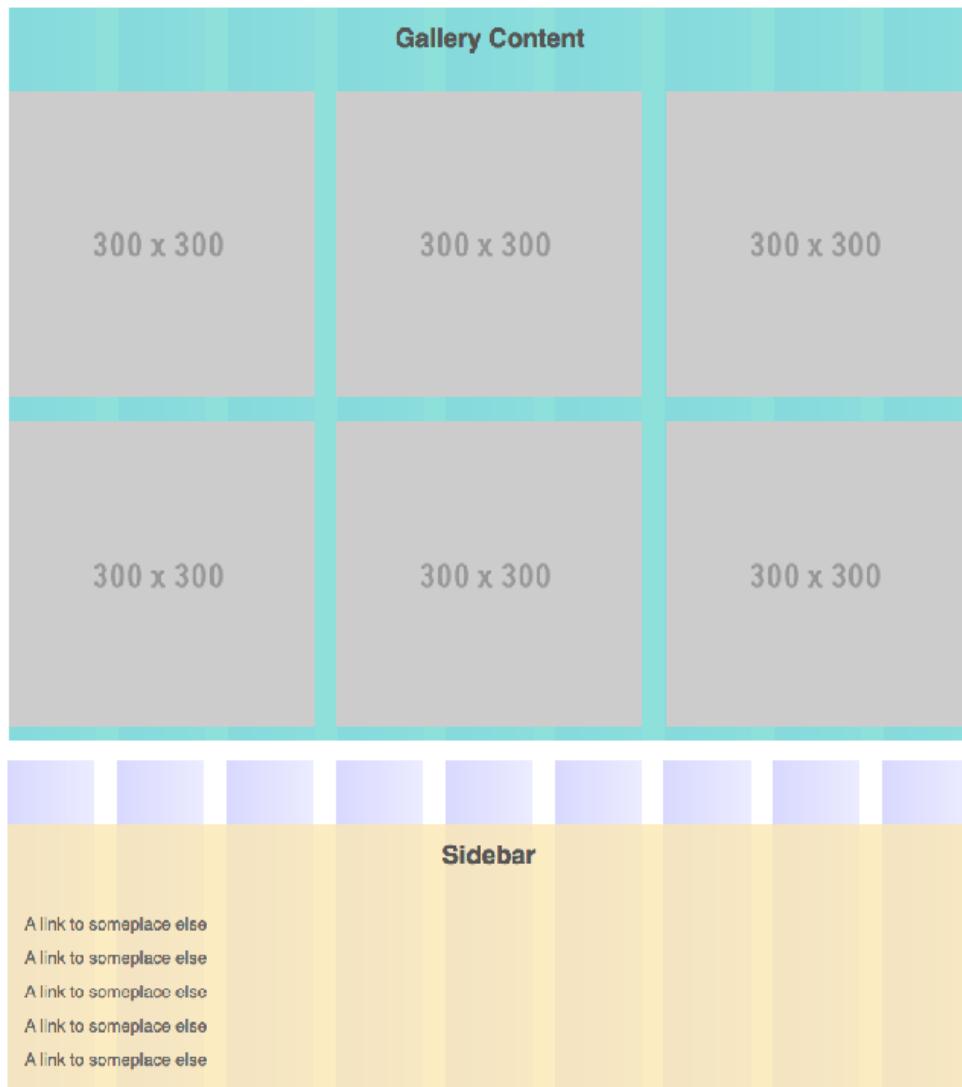
Here are the mobile, tablet and desktop versions of this layout:

Mobile





Tablet



Desktop



Let's start the project by configuring the `$susy` map.

The Susy Map

It's a good practice to code with a mobile-first approach as mentioned previously and we have covered how to do this using `min-width` and `max-width` media queries.

We should also write the `$susy` map and cater it towards the smallest breakpoint on initial load. Hence, we have to set the `columns` setting to 8.

```
// Scss
$susy: (
  columns: 8,
  container: 1140px,
  global-box-sizing: border-box,
  debug: (image: show)
);

@include border-box-sizing;
```

In the spirit of DRYness, let's introduce the use of some Sass variables to hold

our breakpoints since we will be repeating these breakpoints a lot in this chapter. We're naming them `$tablet` and `$desktop` respectively.

```
// Scss
$tablet: 600px;
$desktop: 980px;
```

Now, let's implement what we have learned with the Susy grid background.

Changing Susy's Grid Background For Different Viewports

Import the Breakpoint library if you haven't already done so:

```
@import "breakpoint";
```

If you remember what we did in the previous chapters with `breakpoint()` and `show-grid()`, you should get this easily:

```
// Scss
.wrap {
  @include container();

  // Breakpoint for tablet view
  @include breakpoint($tablet) {
    @include show-grid(9);
  }

  // Breakpoint for desktop view
  @include breakpoint($desktop) {
    @include show-grid(16);
  }
}
```

Bonus points if you are able to use the `susy-breakpoint()` mixin :) Your code should look like this if you used the `susy-breakpoint()` mixin instead:

```
.wrap {  
  @include container();  
  
  // Breakpoint for tablet view  
  @include susy-breakpoint($tablet, 9) {  
    @include show-grid();  
  }  
  
  // Breakpoint for desktop view  
  @include susy-breakpoint($desktop, 16) {  
    @include show-grid();  
  }  
}
```

Try resizing your browser now and you should see the grids changing at the specified breakpoints. :)

One last thing!

It's beneficial to add paddings to the left and right of `.wrap` to ensure that there is sufficient whitespace (say 15px each) to separate our content from the edges of the device.

Since we are using border-box, the actual width of the content will be 1110px (1140px - 15px - 15px) on the largest viewport. If you want to make sure the `max-width` of the inner content is 1140px, you'll have to expand the container to 1170px instead.

```
// Scss
$susy :(
  container: 1170px,
  // Other keys and values
);

.wrap {
  @include container();
  padding-left: 15px;
  padding-right: 15px;

  // Breakpoint for tablet view
  @include breakpoint($tablet) {
    @include show-grid(9);
  }

  // Breakpoint for desktop view
  @include breakpoint($desktop) {
    @include show-grid(16);
  }
}
```

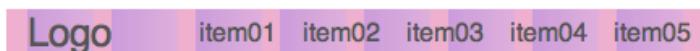
Now that we got the grid background ready for checking our work when working at different viewports, let's move on to making the website proper.

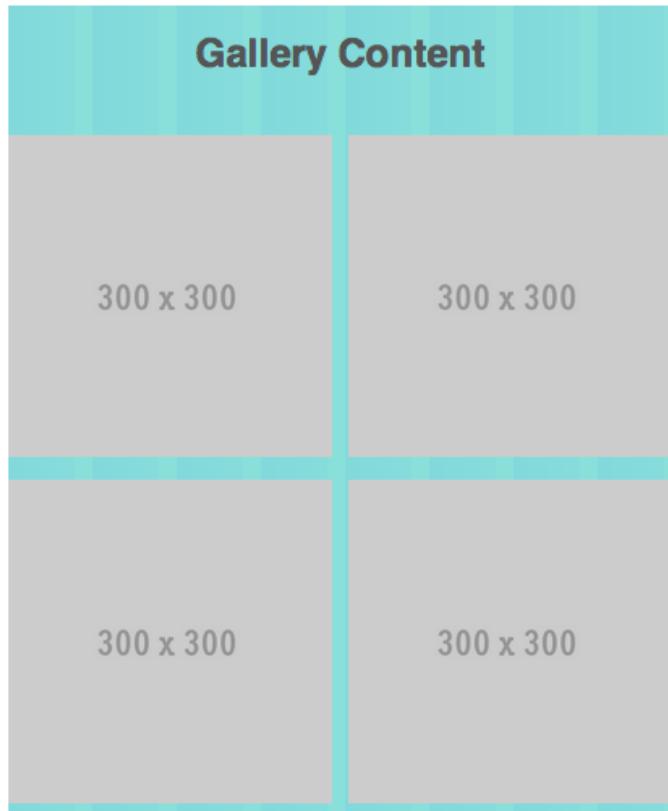
[View Source code \(Tag 12-1\)](#)

The Content and Sidebar Layout

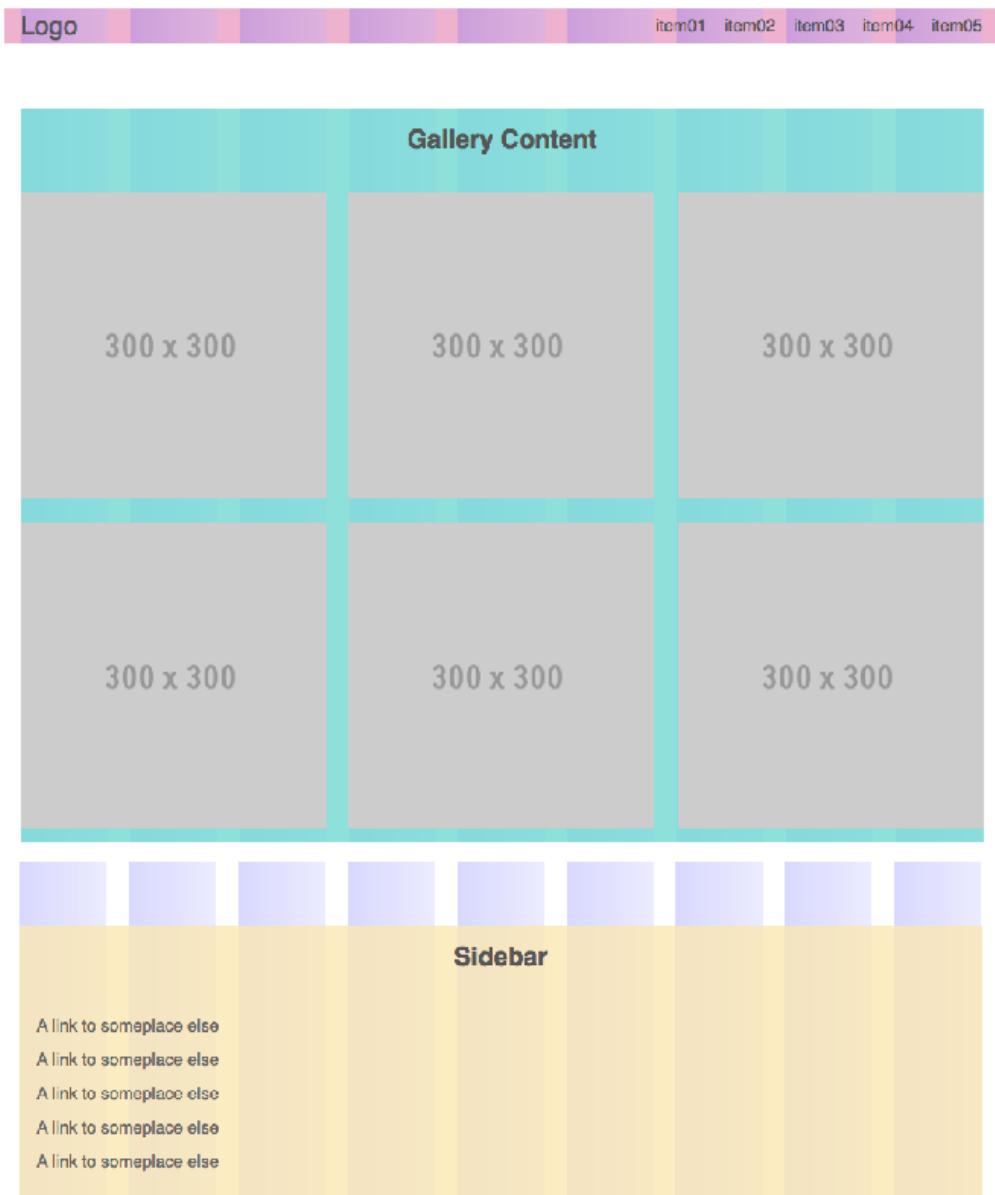
Take note of how the position of `.content` and `.sidebar` changes at different breakpoints:

Mobile

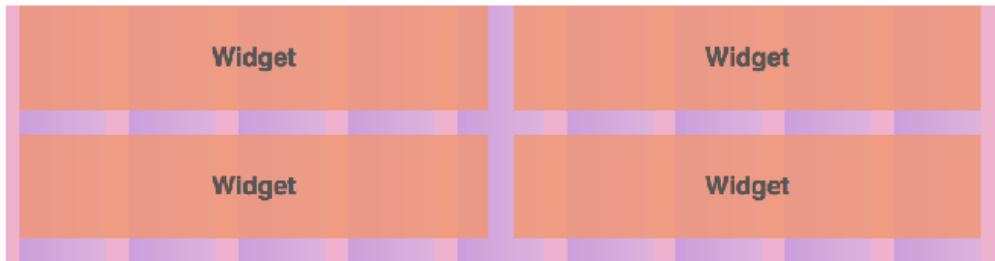




Tablet



Desktop





Notice that `.content` takes up the full 100% width on both mobile and tablet views. But when it goes up to the desktop view, `.content` takes up only 12 of 16 columns.

```
.content {
    // we don't have to do anything for mobile and tablet views
    // because it's set to 100% by default.

    // We only include the breakpoint for desktop view.
    @include susy-breakpoint($desktop, 16) {
        @include span(12);
    }
}
```

Same goes for `.sidebar`, which takes up the full 100% width on the mobile and tablet view. On the desktop view, `.sidebar` takes up 4 of 16 columns, and is the last item in the row.

```
.sidebar {  
  @include susy-breakpoint($desktop, 16) {  
    @include span(4 last);  
  }  
}
```

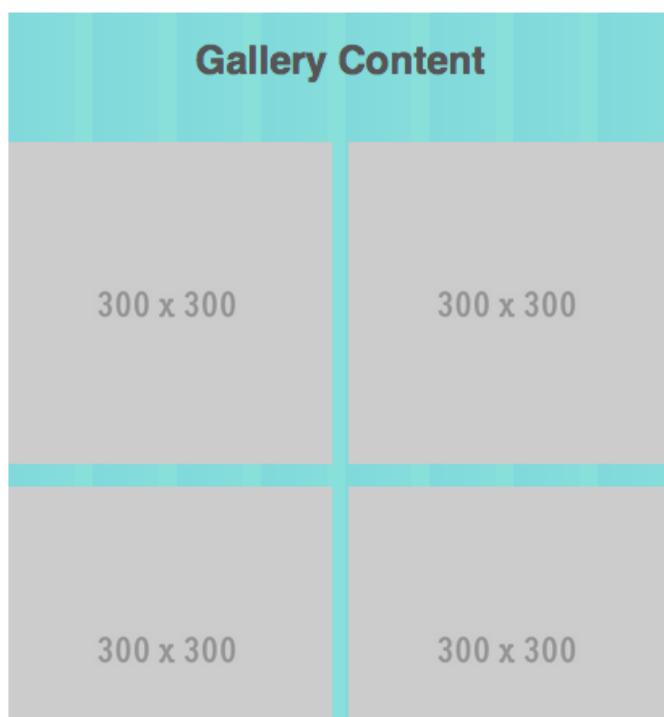
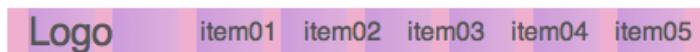
This process repeats for making every part of the website responsive. Now, let's dive into the gallery.

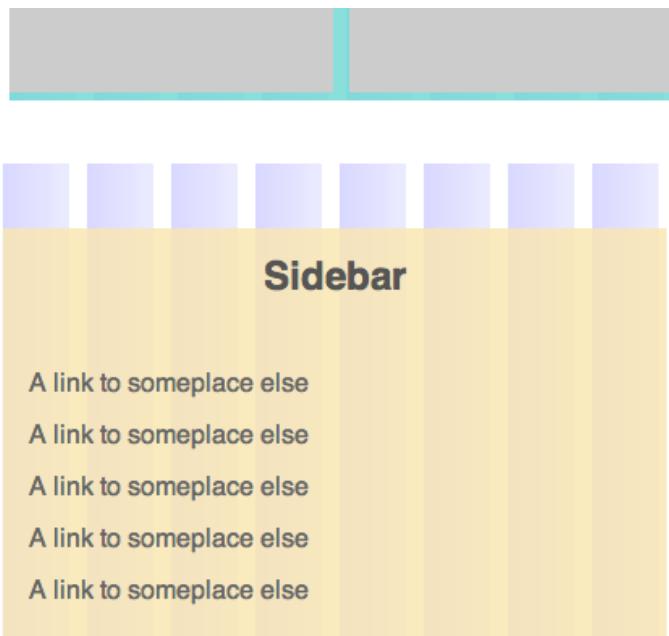
[View Source code](#)

The Gallery

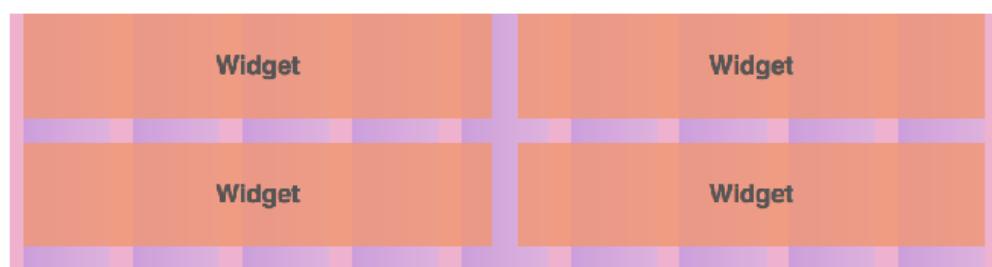
Have a look at the tablet and mobile view layouts again, but this time focus on `.gallery`. Take note of the number of items in each row, and the number of columns they take up.

Mobile





Tablet



We first have to ensure that the gallery takes up the full width on both mobile and tablet views. It shrinks to take up 10 of 12 columns on the desktop view. This can be achieved in the same manner as `.content` and `.sidebar`.

```
.gallery {  
  @include cf;  
  @include susy-breakpoint($desktop, 12) {  
    padding: 0 span(1 wide of 12);  
  }  
}
```

The rest of the gallery is a little tricky because you'll have to use both `min-width` and `max-width` queries. Let's start by using only `min-width` queries to see how it turns out.

On the mobile view, there are two `.gallery__item`s in one row and they both take up 4 of 8 columns. On the tablet view, each row has 3 `.gallery__item`s and they each take up 3 of 9 columns.

This would be the code if we used a pure `min-width` query:

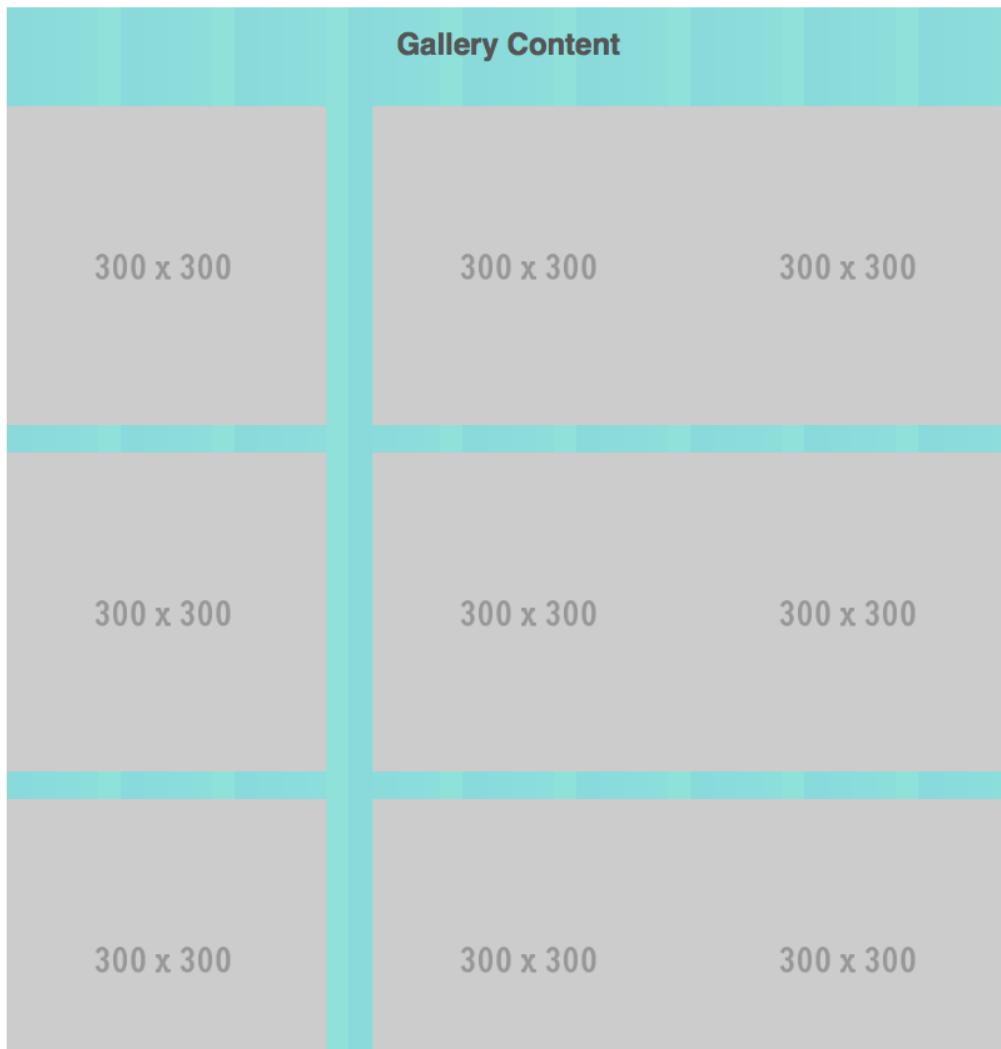
```
.gallery {
  @include cf;
}

.gallery__item {
  @include nested(8) {
    @include span(4);
    margin-bottom: gutter();
    &:nth-child(2n) {
      @include last;
    }
  }
}

@include susy-breakpoint($tablet, 9) {
  @include span(3);
  margin-bottom: gutter();
  &:nth-child(3n) {
    @include last;
  }
}
}
```

The mobile version looks great, but the tablet view doesn't :(

This is what you'll get on the tablet view:



You might have noticed that the item order in the tablet view is messed up too.



This is because the `last()` mixin floats the final item to the right and removes its right margin.

```
// Scss
.last {
  @include last;
}
```

```
/* CSS */
.last {
  float: right;
  margin-right: 0;
}
```

We used this last mixin on every second `.gallery__item` on the mobile view:

```
.gallery__item {
  // ...
  &:nth-child(2n) {
    @include last;
  }
}
```

Adding the `:nth-child()` pseudo class selector increases the CSS specificity, and causes it to be higher than the `.gallery__item` selector we used in the tablet view.

Specificity Calculator

A visual way to understand [CSS specificity](#). Change the selectors or paste in your own.

`.gallery_item`

0

Inline styles

0

IDs

1

Classes, attributes
and pseudo-classes

0

Elements and
pseudo-elements

+ Duplicate

`.gallery_item:nth-child(2n)`

0

Inline styles

0

IDs

2

Classes, attributes
and pseudo-classes

0

Elements and
pseudo-elements

+ Duplicate

What happens then, is that the `float` and `margin` properties of every 2nd item in `.gallery_item` in the mobile view still applies on the tablet view. The mobile view has won the specificity war and has forced its styles onto the tablet view.

One way to handle this is to use a selector with a higher specificity by adding an `:nth-child(2n)` selector into the tablet view as well.

```
.gallery__item {  
  @include susy-breakpoint($tablet, 9) {  
    @include span(3);  
    margin-bottom: gutter();  
  
    // To override mobile view  
    &:nth-child(2n) {  
      @include span(3);  
    }  
  
    &:nth-child(3n) {  
      @include last;  
    }  
  }  
}
```

Unfortunately, this makes code complicated. That's not what we want.

This is where `max-width` queries come into play, and we have spoken about that in the chapter 9 on media queries. Instead of allowing `:nth-child(2n)` to mess around in different viewports, we can restrict this selector to a particular viewport using a combination of `max-width` and `min-width` media arguments.

```
.gallery__item {
  // Constrain span to only mobile
  @include susy-breakpoint(max-width $tablet, 8) {
    @include span(4);
    margin-bottom: gutter();
    &:nth-child(2n) {
      @include last;
    }
  }

  // Constrain span to only tablet
  @include susy-breakpoint($tablet $desktop, 9) {
    @include span(3);
    margin-bottom: gutter();
    &:nth-child(3n) {
      @include last;
    }
  }
}
```

Of course, we can now add the desktop view into the mix:

```
// Scss
.gallery__item {
    // Constrain span to only mobile
    @include susy-breakpoint(max-width $tablet, 8) {
        @include span(4);
        margin-bottom: gutter();
        &:nth-child(2n) {
            @include last;
        }
    }

    // Constrain span to only tablet
    @include susy-breakpoint($tablet $desktop, 9) {
        @include span(3);
        margin-bottom: gutter();
        &:nth-child(3n) {
            @include last;
        }
    }

    // Constrain span to only desktop
    @include susy-breakpoint($desktop, 10) {
        @include span(2);
        margin-bottom: gutter();
        &:nth-child(5n) {
            @include last;
        }
    }
}
```

We're complete with the gallery! Next up, the footer!

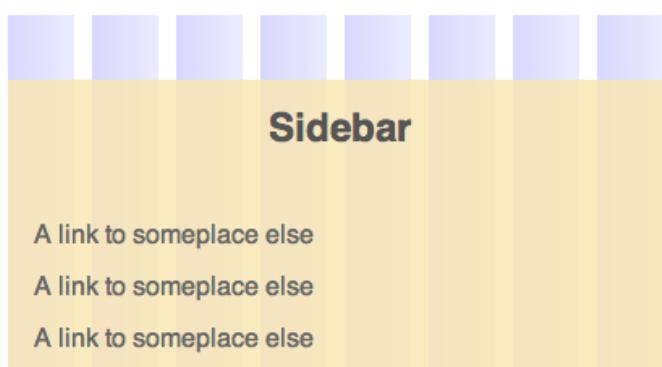
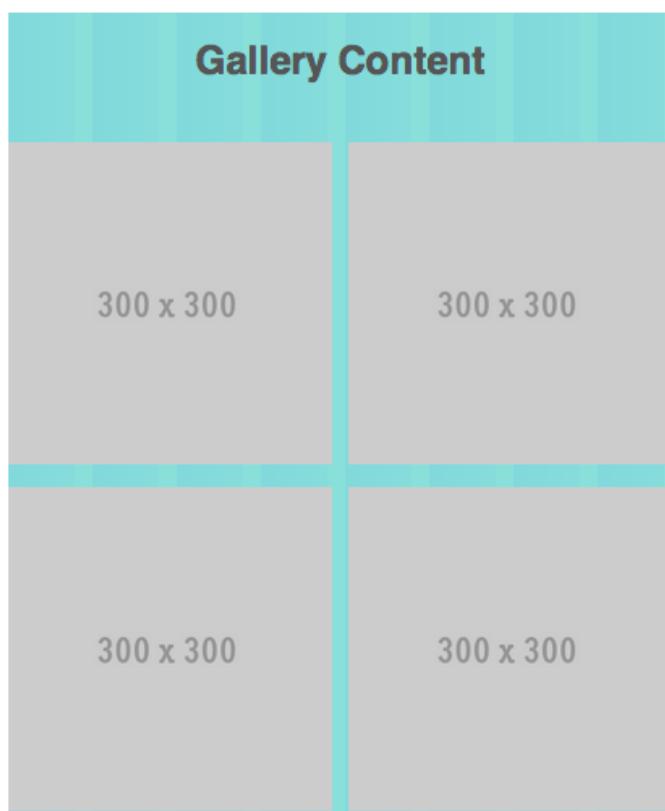
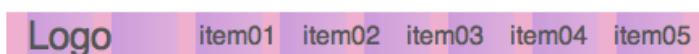
[View Source code](#)

Footer

The `.widgets` within the footer can be coded the same way as `.gallery__item` if you chose to use the `span()` mixin to create it. Let's look at the difference in the Sass code we have to use since we went with the `gallery()` mixin instead.

Once again, here's how each of the 3 layouts should look like. Focus on the footer this time:

Mobile



A link to someplace else

A link to someplace else

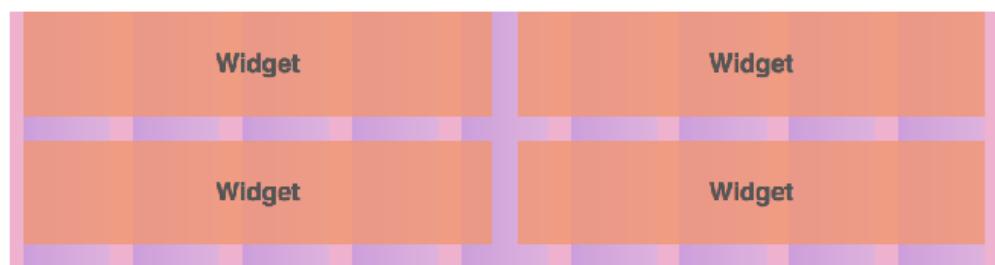
Widget

Widget

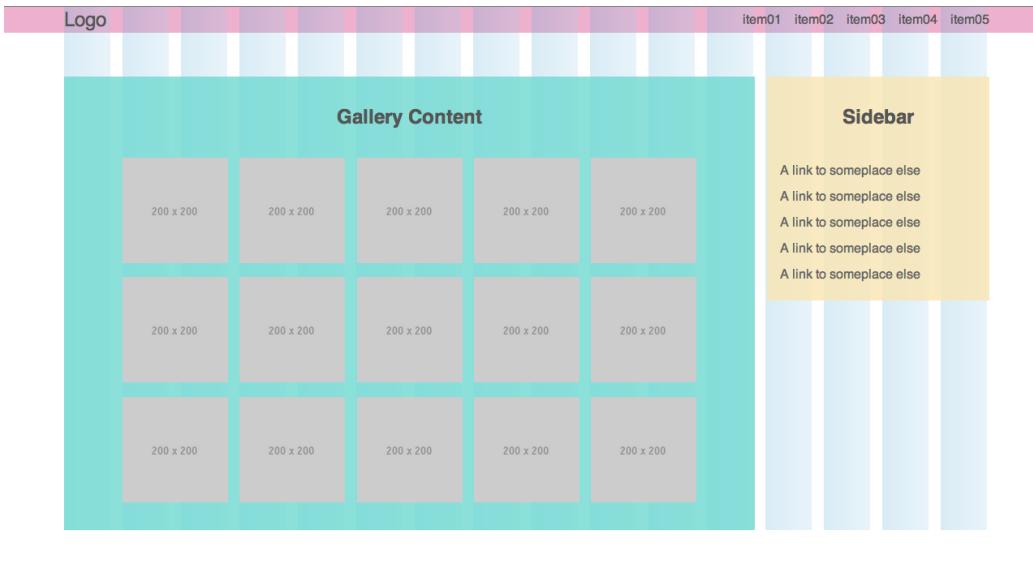
Widget

Widget

Tablet



Desktop



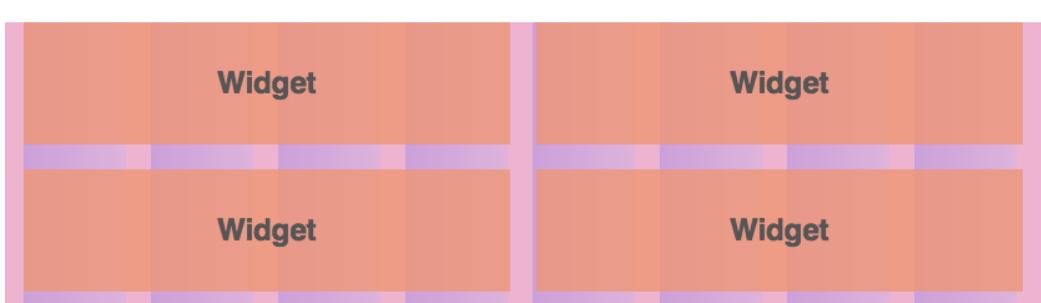
There are two `.widgets` on each row, and each widget takes up 4 of 8 columns on both the mobile view and the tablet view.

```
.widget {
  @include gallery(4);
  margin-bottom: gutter();
}
```

Even though there are supposed to be 9 columns as the background grid suggests, Susy allows you to change this context to any number of columns you want. The `.widget`s here on a tablet view are a prime example of this.

The `.widget`s don't align to the background grid because its context is set to 8 columns while the context of the background grid is set to 9 columns.

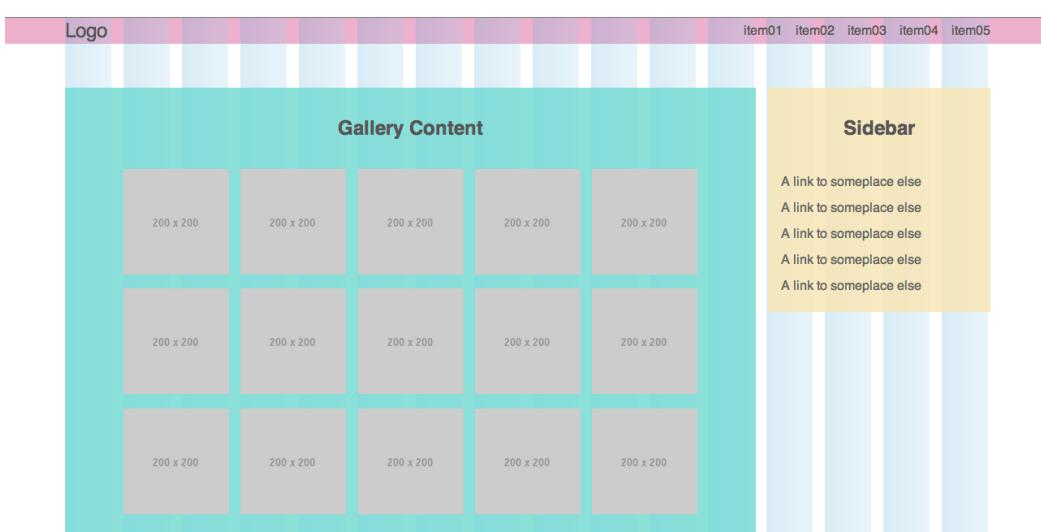
If you switch the context of the background grid to 8 columns, the `.widget`s will align perfectly.



The layout will then break at the desktop width and each `.widget` will take up 4 of 16 columns.

```
.widget {
  @include gallery(4);
  margin-bottom: gutter();

  @include susy-breakpoint($desktop, 16) {
    @include gallery(4);
    margin-bottom: 0;
  }
}
```



Using the gallery is a lot simpler compared to the `span()` mixin when doing responsive design because we don't have to wrangle with media queries!

The reason this works is because the CSS output from the `gallery()` mixin adds a `:nth-child()` selector to each of the elements.

```
/* CSS */
.widget {
    /* some props */
}
.widget:nth-child(2n + 1) {
    /* some props */
}
.widget:nth-child(2n + 2) {
    /* some props */
}
@media (min-width: 980px) {
    .widget {
        /* some props */
    }
    .widget:nth-child(4n + 1) {
        /* some props */
    }
    .widget:nth-child(4n + 2) {
        /* some props */
    }
    .widget:nth-child(4n + 3) {
        /* some props */
    }
    .widget:nth-child(4n + 4) {
        /* some props */
    }
}
```

This `.widget:nth-child()` selector within the media query has the same specificity as the `.widget:nth-child()` without media queries. Since there is a tie in the specificity war, the one lower down the stylesheet will take effect and we get the styles we wanted without having to wrangle with the breakpoints.

A Quick Wrap Up

We have covered a lot of ground in this chapter, and it'll be good to quickly summarize what we have done.

In this chapter, you have learned how to apply theories that were introduced in the last 3 chapters. Specifically, you now know how to build responsive websites using Susy with both the `susy-breakpoint()` and `breakpoint()` mixins.

You also had a real taste of how to use both `min-width` and `max-width` media queries to simplify your code.

Finally, we explored how to write a responsive layout with Susy using both the `span()` and `gallery()` mixins.

You're now well on your way to move onto the infinite possibilities that Susy provides us with. In the next chapter, we are going to dive further into the different Susy gutter settings and see how they affect your Susy code.

Understanding Gutter Positions

Susy comes with 5 different gutter settings that you can use to create grids in variety of ways.

Each gutter position affects how you code with Susy, and they provide you with the flexibility to change how your grid is coded.

So far, we have only talked about the `after` gutter position. We will be exploring the rest of the gutter positions in this chapter.

You'll learn:

- How to use the `after`, `before`, `split`, `inside` and `inside-static` gutter positions
- What are the differences and nuances between each gutter position

We will create the same layout using all 5 gutter position settings so you have a base to compare between the settings.

We'll start off with the `after` option since that's the one we've used previously, followed by `before`, `split`, `inside` and `inside-static`.

This is the layout we will make in this chapter:



Taking care of CSS

Here's the full CSS for every item we will need.

```
.content {
  margin-top: 10vh;
  background: rgba(113, 218, 210, 0.5);
}

.sidebar {
  margin-top: 10vh;
  height: 40vh;
  background: rgba(250, 231, 179, 0.8);
}

.level-1:nth-child(odd){
  background: rgba(244, 49, 11, 0.5);
}

.level-1:nth-child(even){
  background: rgba(250, 231, 179, 0.5);
}

.level-2:nth-child(odd){
  background: rgba(20, 231, 179, 0.5);
}

.level-2:nth-child(even){
  background: rgba(124, 11, 19, 0.5);
}

.full {
  margin-top: 1rem;
  background: rgba(123, 222, 22, 0.5);
}

h2 {
  padding: 1rem 0;
  text-align: center;
  color: #555;
}
```

Also, do not forget about the clearfix mixin.

```
@mixin cf {  
  &:after {  
    content: " ";  
    display: block;  
    clear: both;  
  }  
}
```

Writing the HTML

The HTML this time round is more complex compared to the previous chapters because we are nesting elements two levels down. Let's walk through the HTML together.

First, we begin with a `.content` and a `.sidebar` div within a `.wrap` container just like earlier examples.

```
<div class="wrap">  
  <div class="content"><h2>Content</h2></div>  
  <div class="sidebar"><h2>Sidebar</h2></div>  
</div>
```

Within the content area, we have a gallery-like content with only 3 items in a row. Let's call it the `.gallery` since it looks like one. Within gallery, there are 3 `gallery_items`.

```
<div class="wrap">

  <div class="content">
    <h2>Content</h2>
    <div class="gallery">
      <div class="gallery__item"><h2>Item 1</h2></div>
      <div class="gallery__item"><h2>Item 2</h2></div>
      <div class="gallery__item"><h2>Item 3</h2></div>
    </div>
  </div>

  <div class="sidebar"><h2>Sidebar</h2></div>
</div>
```

Gallery items (1) and (2) have two more items within them, let's call those

.gallery__subitem .

```
<div class=".gallery">
  <!-- Item 1-->
  <div class="gallery__item">
    <h2>Item 1</h2>
    <div class="gallery__subitem">
      <h2>Subitem 1</h2>
    </div>
    <div class="gallery__subitem">
      <h2>Subitem 2</h2>
    </div>
  </div>
  <!-- Item 2 -->
  <div class="gallery__item">
    <h2>Item 2</h2>
    <div class="gallery__subitem">
      <h2>Subitem 3</h2>
    </div>
    <div class="gallery__subitem">
      <h2>Subitem 4</h2>
    </div>
  </div>
  <!-- Item 3 -->
  <div class="gallery__item">
    <h2>Item 3</h2>
  </div>
</div>
```

Finally, after `.gallery`, we have another div within the content. Let's call that `.full`.

```
<div class="wrap">

  <div class="content">
    <h2>Content</h2>
    <div class="gallery">
      <!-- gallery stuff -->
    </div>
    <div class="full"><h2>Full Wide Item</h2></div>
  </div>

  <div class="sidebar"><h2>Sidebar</h2></div>
</div>
```

The Susy Map

As usual, we start off by configuring the Susy map.

```
$susy :(
  columns: 8,
  container: 1140px,
  global-box-sizing: border-box
  debug: (image: show)
);

@include border-box-sizing;

.wrap {
  @include container;
}
```

Gutter-position: After

The gutter position can be configured in the Susy map with the `gutter-position` key.

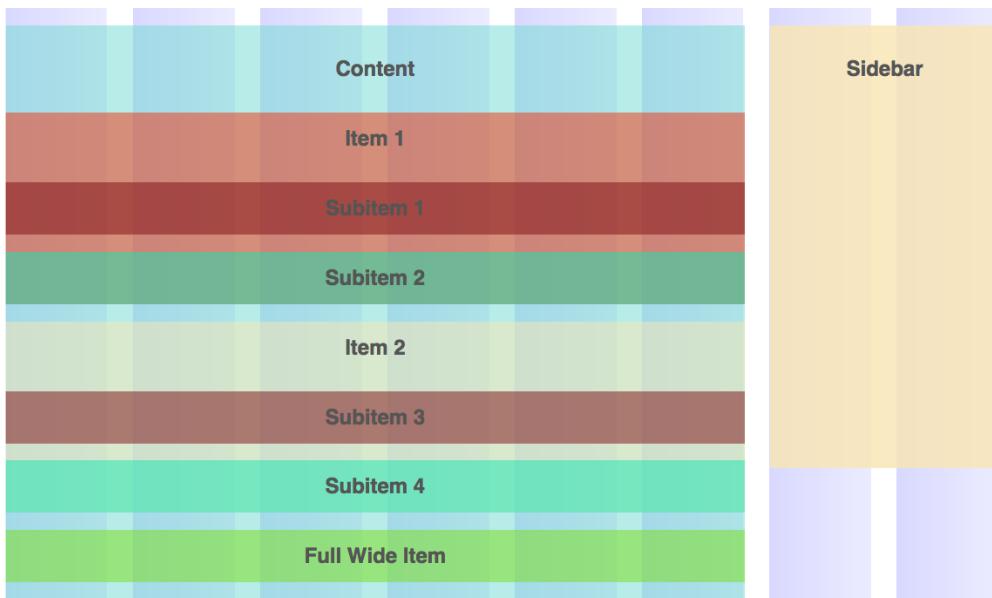
```
$susy :(  
  //...  
  gutter-position: after,  
)
```

Let's take a look at our target layout again.



We know that the first thing we need to build is the `.content` and `.sidebar` layout. `.content` takes up 6 columns while `.sidebar` takes up 2 columns:

```
// SCSS  
.content {  
  @include span(6);  
}  
  
.sidebar {  
  @include span(2 last);  
}
```



The `.gallery` contains `.gallery__items` which are all going to be floated and we need to give it a clearfix.

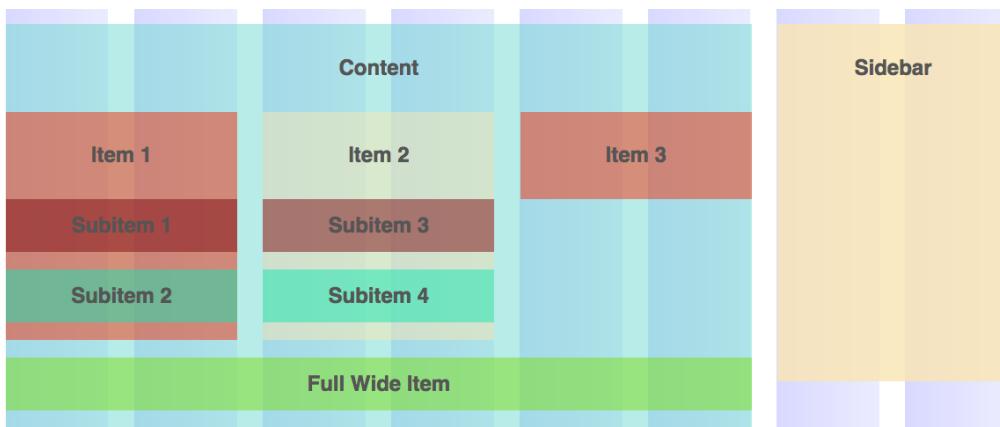
```
// Scss
.gallery {
  @include cf;
}
```

Items 1, 2 and 3 each take up 2 of 6 columns. We can make this either by using the `gallery()` or the `span()` mixin.

```
.gallery__item {
  @include span(2 of 6);
  &:last-child {
    @include last;
  }
}
```

Since everything within content has a nested context of 6 columns, we can optionally wrap everything within a `nested()` mixin:

```
.gallery__item {  
  @include nested(6) {  
    @include span(2);  
    &:last-child {  
      @include last;  
    }  
  }  
}
```



Each sub-item takes up 1 of 2 columns each. The last item on each row must have the `last` argument as well.

```
.gallery__subitem {  
  @include nested(2) {  
    @include span(1);  
    &:last-child {  
      @include last;  
    }  
  }  
}
```

Since everything within `.gallery__item` is floated, we have to give `.gallery__item` a clearfix.

```
.gallery__item {  
  @include cf;  
}
```

That's how we get the layout we want with the `after` gutter position setting.

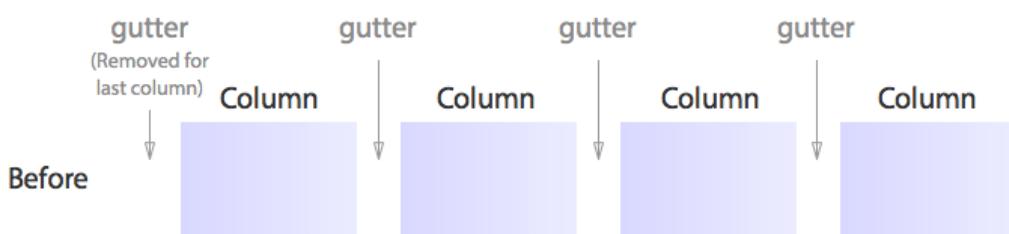


[View Source code](#)

Let's try building the same thing with `gutter-position: before`.

Gutter-position: Before

`before` is very similar to `after`. Susy now adds the gutters before each column instead.



To use `before`, we will have to change the `$susy` map slightly:

```
$susy: (
  // ...
  gutter-position: before
);
```

Since Susy now adds the margin before every element, we have to remove the gutter on the very first item of the row instead.

We can either do this with the `first` argument, or the `first` mixin. Either of these will force `margin-left` to 0.

```
// Scss
.first-mixin {
  @include first;
}

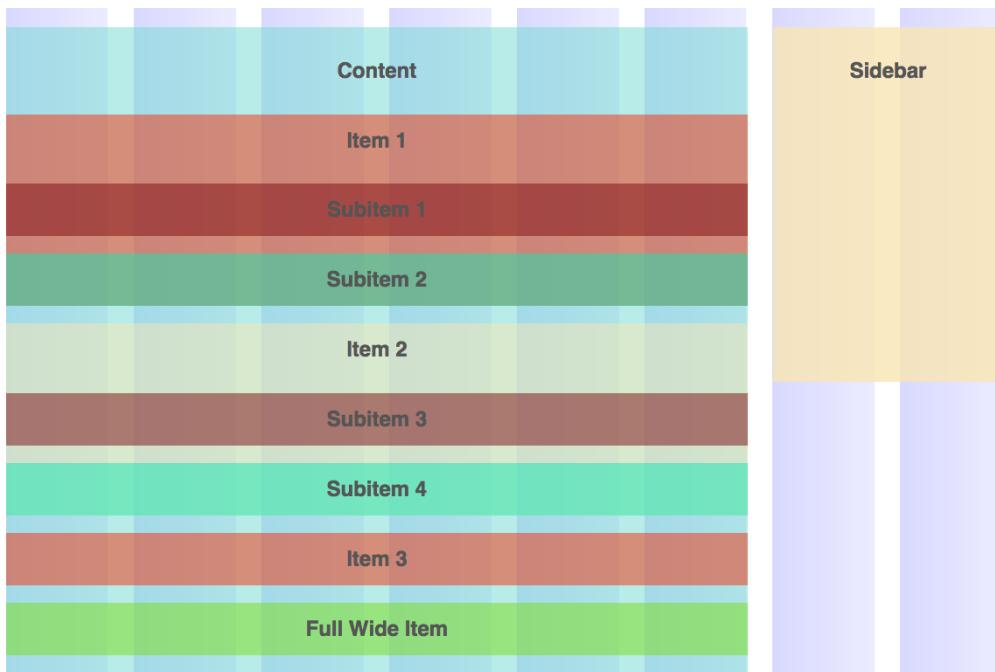
.first-keyword {
  @include span(2 of 8 first);
}
```

Let's start styling!

`.content` takes up 6 of 8 columns and `.sidebar` takes up 2 of 8 columns. Since `.content` is the first item of the row, we have to give it a `first` keyword or the `first()` mixin.

```
// Scss
.content {
  @include span(6 first);
}

.sidebar {
  @include span(2);
```

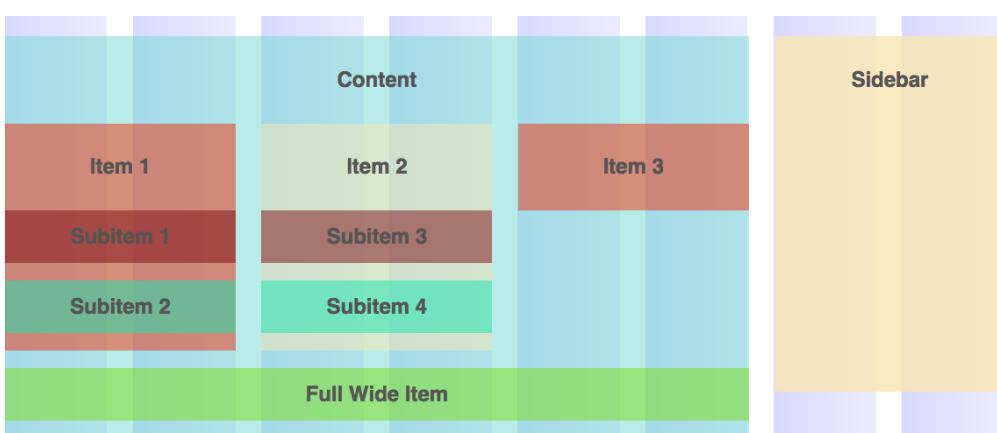


Next, the `.gallery` requires a clearfix because all of its child elements are going to be floated.

```
// Scss
.gallery {
  @include cf;
}
```

Each `.gallery__item` takes up 2 of 6 columns each and we have to give Item 1 a `first` keyword or the `first()` mixin. We also have to give each `.gallery__item` aclearfix since all content within `.gallery__item` is floated.

```
// Scss
.gallery__item {
  @include cf;
  @include nested(6) {
    @include span(2);
    &:first-child {
      @include first;
    }
  }
}
```



Each `.gallery__subitem` takes up 1 of 2 columns. We also need to give every first sub-item a `first()` mixin. However, because the `:first-child` in the html is a `<h2>` element, we will have to use the `nth-child()` selector to target the sub-item correctly.

```
// Scss
.gallery__subitem {
  @include nested(2) {
    @include span(1);
    &:nth-child(2) {
      @include first;
    }
  }
}
```

And we're done with creating the exact same layout using `before` as the gutter-position !

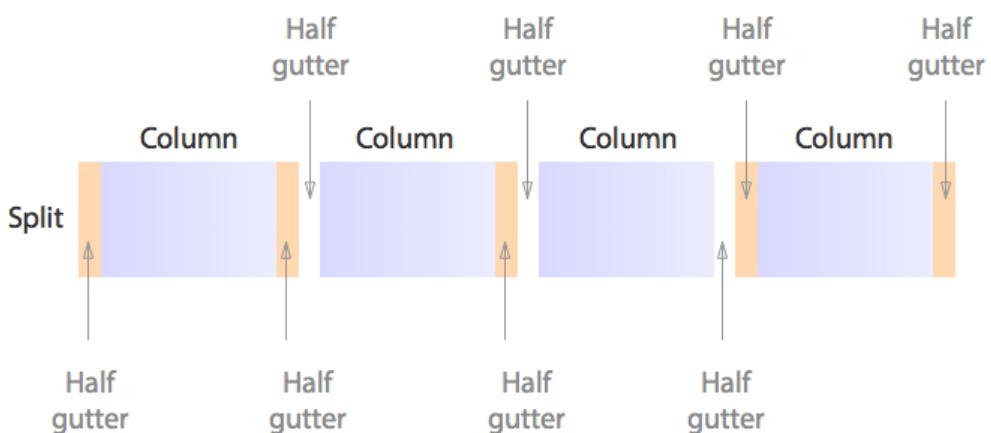


Some people prefer to use `before` as opposed to `after` because you can now create the Susy layout by using the `:first-child()` selector most of the time. This means that you can provide support to IE8 without the use of polyfills.

[View Source code](#)

Gutter-position: Split

`split` works differently from both `before` and `after`. In `split`, the gutters are divided in two and placed on both sides of the column.



There are two fundamental differences when using the `split` position compared to both `before` and `after`.

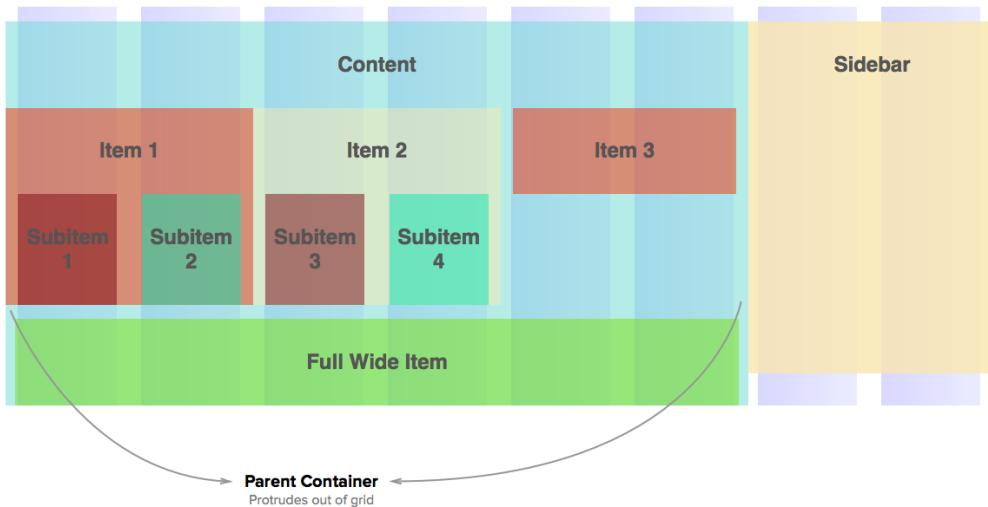
1. There is no need to remove the gutters from the extreme edges of the row.
2. Parent elements must contain a `nest` argument.

When we use `before` and `after`, the width of the element consists of 8 columns and 7 gutters. When working with `split` however, the width of the elements consists of 8 columns and 8 gutters instead.

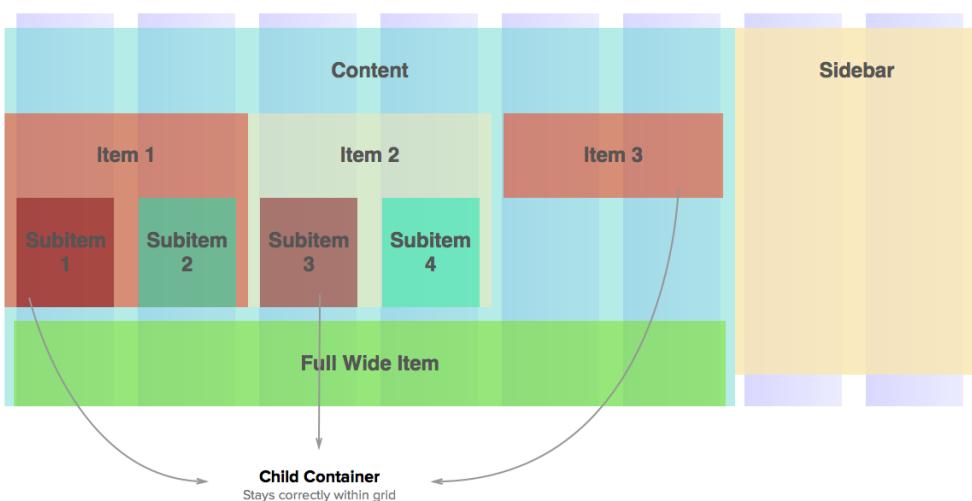
Because of this fundamental difference, the grid background and grids will look a little bit different from that of `before` and `after`.



If we take a closer look at the grid, we can see that each parent element seems to be protruding out of alignment on the grid. You will also notice that the extreme left and right edges of the grids are gutter spaces instead of column spaces.



Furthermore, we also see that the child elements are the ones that ultimately align nicely with the grid.



Parent elements are external containers that help with the process of aligning the final child element on the grid. You have to know how to identify parent and child elements and give the parent the `nest` keyword.

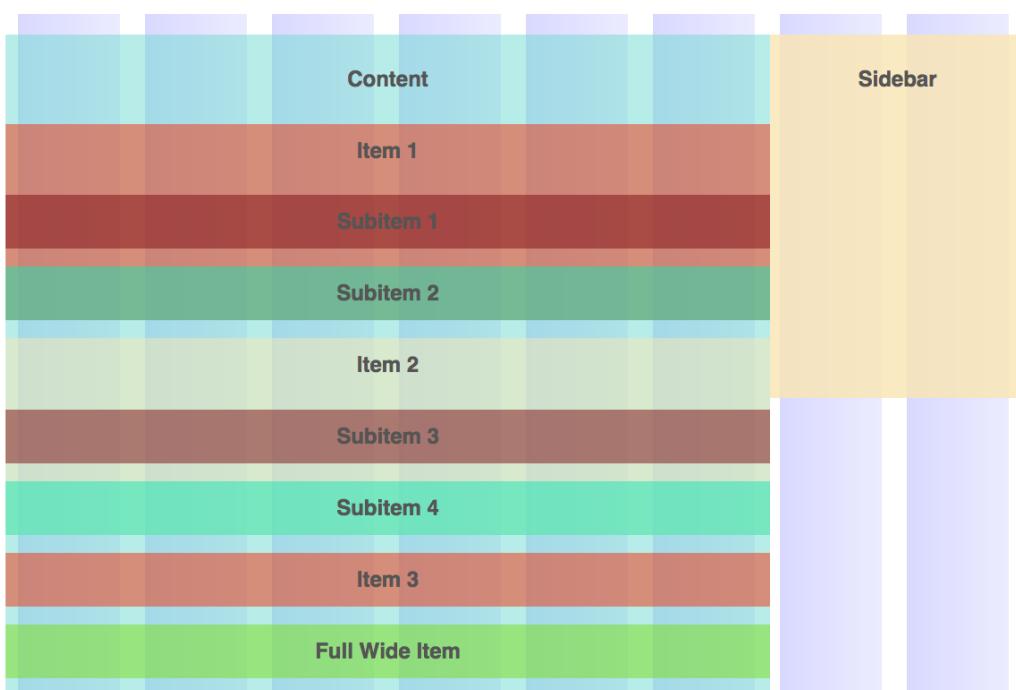
Let's start styling!

`.content` takes up 6 of 8 columns while `.sidebar` takes up 2 of 8 columns. Both `.content` and `.sidebar` need a `nest` keyword because they contain child elements that we are going to span with Susy.

```
// SCSS
.content {
  @include span(6 of 8 nest);
}

.sidebar {
  @include span(2 of 8 nest); // We are assuming .sidebar
contains spanned child elements
}
```

Note: The `last` keyword is optional when using Split because everything can be floated left and the row will be okay. You can optionally apply the `last` keyword here to float the `.sidebar` right to combat against subpixel rounding errors if you wish to.



Next, the `.gallery` requires a clearfix because all of its child elements are going to be floated.

```
// Scss
.gallery {
  @include cf;
}
```

Each `.gallery__item` takes up 2 of 6 columns. We have to give items 1 and 2 a `nest` keyword because they contain elements that we are using the `span()` mixin on. Item 3 is not given the `nest` keyword because there are no elements that we are will be spanning with Susy within it.

```
// Scss
.gallery__item {
  @include cf;
  @include nested(6) {
    @include span(2 nest);
    &:nth-child(3) {
      @include span(2)
    }
  }
}
```



Each `.gallery__subitem` takes up 1 of 2 columns.

```
// Scss
.gallery__subitem {
  @include nested(2) {
    @include span(1);
  }
}
```



There's just one more thing to take note of. `.full` is also a child element. We need to give it a span to ensure that it doesn't protrude out of the grid.

```
.full {
  @include span(full);
}
```

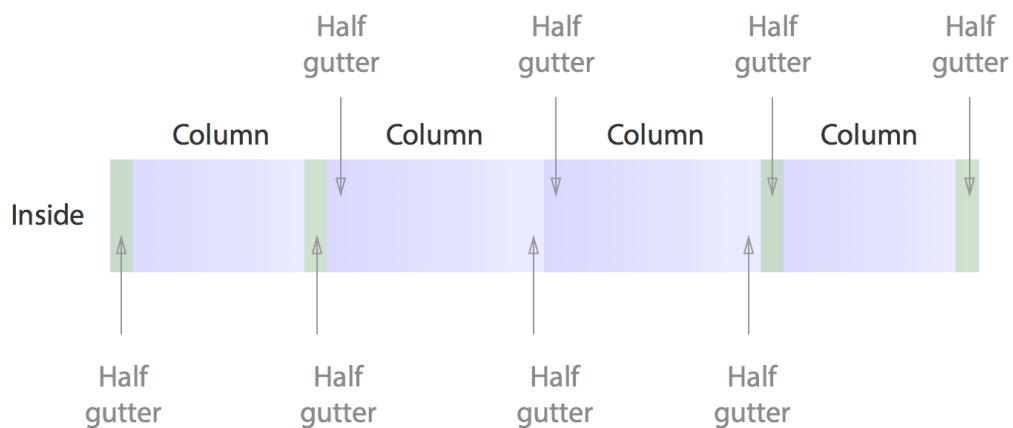
And we're done.



[View Source code](#)

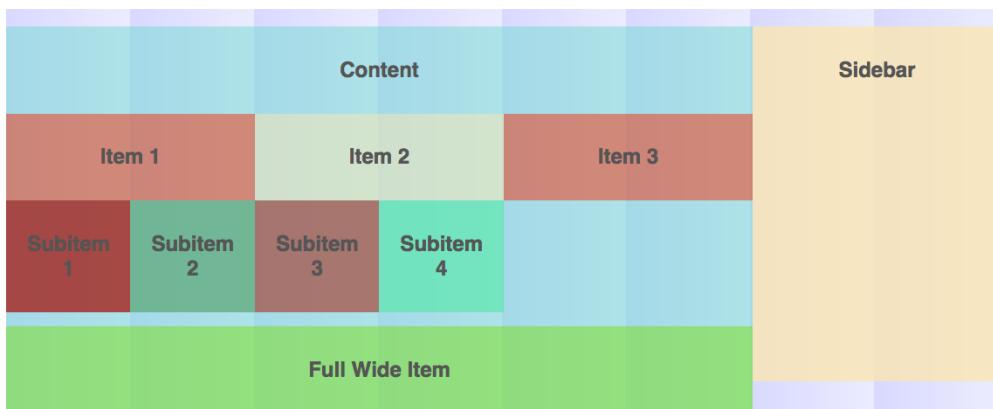
Gutter-position: Inside

`inside` and `split` are similar. The difference is that `inside` outputs gutters as `padding` instead of `margin`.

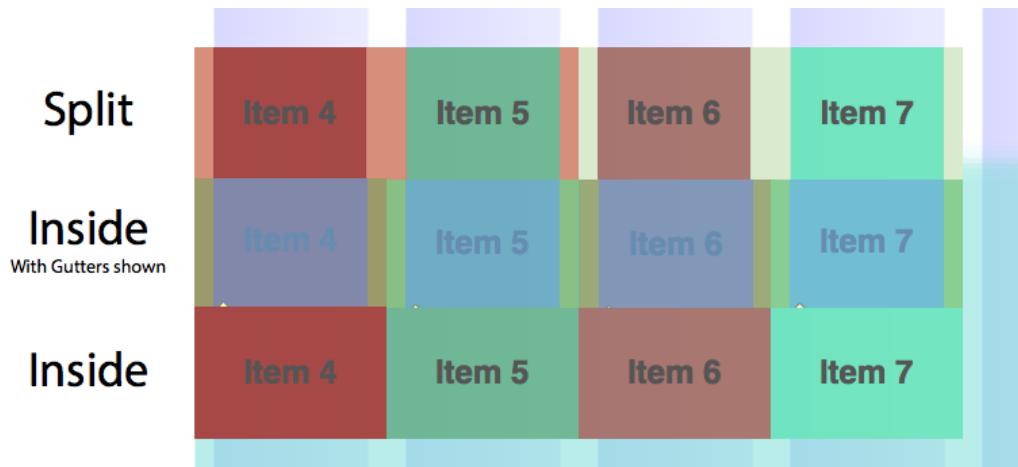


Like `split`, you need to be able to identify the parent elements and give them the `nest` argument.

Because `padding` is used instead of `margin`, the background grid will show up very differently with `inside` when compared to `split`.



The gutters are actually accounted for within the columns, so you can't see them on this grid. Take a look at the diagram below, which illustrates the difference between the grids generated by `split` and `inside`.



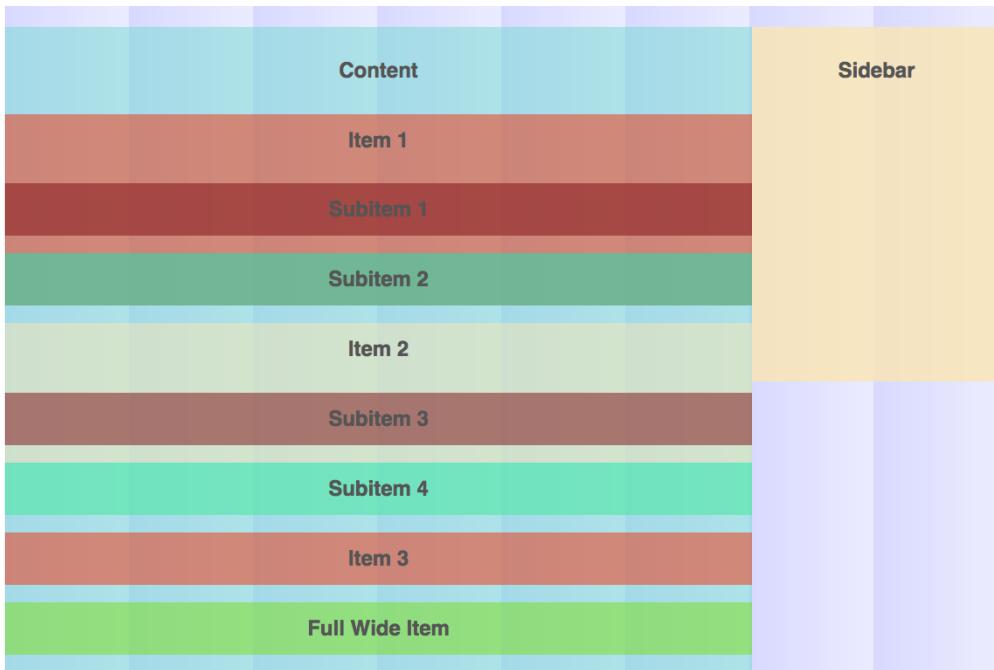
If we hover over an item that is created with `inside`, you can see that the width of the content is exactly the same as that of the `split` background grid. `inside` is hence identical to the `split` grid with the exception that the gutters are made with `padding`.

Let's start styling!

The code will be exactly the same as what we did for `split`. Again, `.content` takes up 6 of 8 columns while `.sidebar` takes up 2 of 8 columns. Both `.content` and `.sidebar` need a `nest` keyword because they contain child elements that we are going to span with Susy.

```
// SCSS
.content {
  @include span(6 of 8 nest);
}

.sidebar {
  @include span(2 of 8 nest); // We are assuming `sidebar` contains spanned child elements
}
```

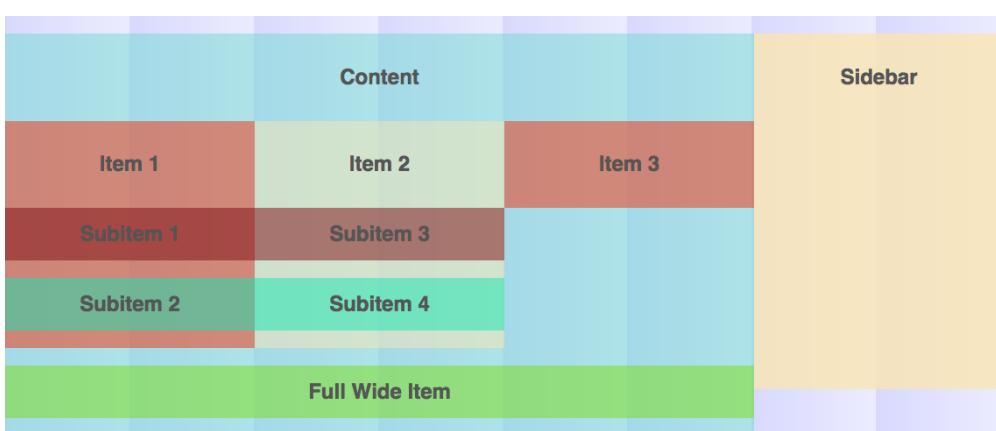


Next, `.gallery` requires a clearfix because all of its child elements are going to be floated.

```
// Scss
.gallery {
  @include cf;
}
```

Each `.gallery__item` takes up 2 of 6 columns. We have to give items 1 and 2 a `nest` keyword because there are elements within that we have to span. Item 3 is not given the nest keyword because there are no elements that we will be spanning with Susy within it.

```
// Scss
.gallery__item {
  @include cf;
  @include nested(6) {
    @include span(2 nest);
    &:nth-child(3) {
      @include span(2)
    }
  }
}
```



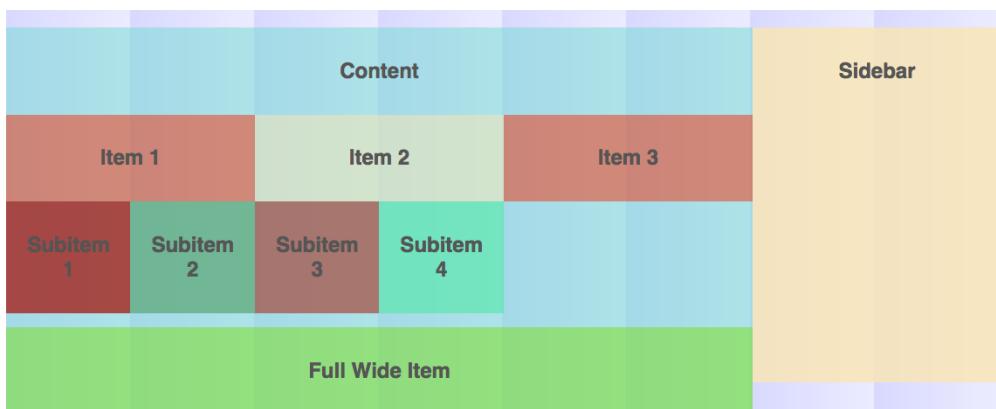
Each `.gallery__subitem` takes up 1 of 2 columns.

```
// Scss
.gallery__subitem {
  @include nested(2) {
    @include span(1);
  }
}
```



Finally, we have to tell Susy that `.full` is also a child element.

```
.full {
  @include span(full);
}
```



And that's how you layout grids with `inside`.

There is one more quirk with `inside` that is different from the rest of the layouts – `border-box` is required for `inside`.

If you have left `global-box-sizing` as `content-box` in the `$susy` map, Susy will automatically create the `border-box` property locally for every single span you use.

```
/* CSS */
.inside-span {
  -moz-box-sizing: border-box;
  -webkit-box-sizing: border-box;
  box-sizing: border-box;
  width: 75%;
  float: left;
  padding-left: 1.25%;
  padding-right: 1.25%;
}
```

So if you're going to use `inside`, use `border-box` to reduce the number of properties declared.

Finally, `inside` and `inside-static` are used in the same way. The difference between them is that `inside-static` requires `column-width` to be declared in the `$susy` map, and gutters will be output as static units instead of percentages.

[View Source code](#)

A Quick Wrap Up

We have covered the 5 different gutter positions in Susy and how to use them. It's worth revisiting this chapter to fully understand how to use the different gutter positions so you may use them fluently when the situation calls for it.

Here's a very quick summary of the 5 gutter positions:

After

- Insert `last` keyword or `last()` mixin to the last item of every row

Before

- Insert `first` keyword or `first()` mixin to the first item of every row

Split

- No need to remove any gutters
- Parent elements must have a `nest` keyword

Inside

- No need to remove any gutters
- Parent elements must have a `nest` keyword
- Gutters are output as padding

Inside-static

- No need to remove any gutters
- Parent elements must have a `nest` keyword
- Gutters are output as padding
- Static units for gutters can be used if a `column-width` key is set on the `$susy` map.

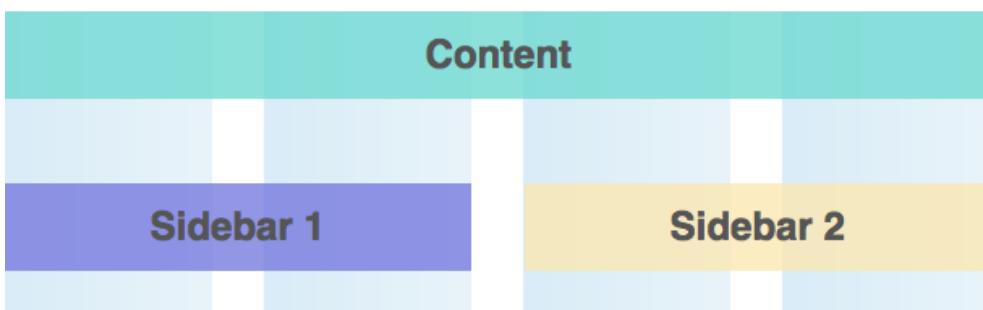
Next, we are going to find out how to use the Isolate Technique to place elements in their correct positions and mitigate all subpixel rounding errors.

The Isolate Technique

Isolate is a layout technique that was initially created to address sub-pixel rounding errors by browsers. They can also be used to create layouts that normal floats cannot. We will be building one such layout in this chapter.

This layout has two views.

Mobile view:



Desktop view:



Notice how `.content` is the topmost element on the mobile and the center element on the desktop? This is something that normal floats can't do.

So in this chapter, you'll learn:

- How isolate works
- How to use the 3 available isolate methods in Susy layouts

Before we begin, let's set up the HTML and CSS files.

Taking Care of CSS

As usual, here is the CSS we need for this chapter.

```
// Scss. Taking Care of CSS
.content {
  margin-top: 5vh;
  background: rgba(113, 218, 210, 0.8);
}

.content-2 {
  margin-top: 5vh;
  background: rgba(113, 218, 210, 0.8);
}

.sidebar2 {
  margin-top: 5vh;
  margin-bottom: 5vh;
  background: rgba(250, 231, 179, 0.8);
}

.sidebar1 {
  margin-top: 5vh;
  margin-bottom: 5vh;
  background: rgba(120, 122, 222, 0.8);
}

h2 {
  padding: 2.5vh 0;
  text-align: center;
  color: #555;
}
```

Writing the HTML

The HTML can be a little tricky here. We have to make sure `.content` is the top most element followed by `.sidebar1` and `.sidebar2`.

This is because we are coding from a mobile-first approach and we would want content to come first.

This HTML order is important because we can change the location of `.content` horizontally on the desktop but not the vertical arrangement of divs on mobile.

```
<!-- HTML -->
<div class="wrap">

    <div class="content">
        <h2>Content</h2>
    </div>

    <div class="sidebar1">
        <h2>Sidebar 1</h2>
    </div>

    <div class="sidebar2">
        <h2>Sidebar 2</h2>
    </div>

</div>
```

The Susy Map

We are going to set `columns` to 4 since the mobile view contains a 4-column grid and we want to build the layout using a mobile-first approach. The other settings are the same as what we used for earlier layouts.

```
// Scss
$susy: (
  columns: 4,
  container: 1140px,
  global-box-sizing: border-box,
  debug: (image: show)
);

@include border-box-sizing;

.wrap {
  @include container;
}
```

We're set to begin coding for this chapter. To fully understand the code that Susy provide us with, we need to develop a clear understanding of how the isolate technique works. Let's discuss that first.

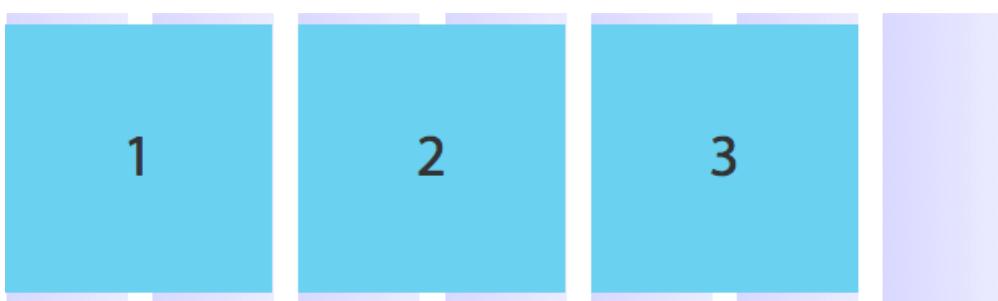
The Isolate Technique

The isolate technique relies on negative margins to push and pull elements from the left to ensure that they are placed in their proper positions. Here's how the code of an isolated element would look like:

```
/* CSS */
.widget {
  width: 24.05063%;
  float: left;
}
.widget:nth-child(4n + 1) {
  margin-left: 0;
  margin-right: -100%;
  clear: both;
}
.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;
}
```

Let's explain the isolate technique with an example.

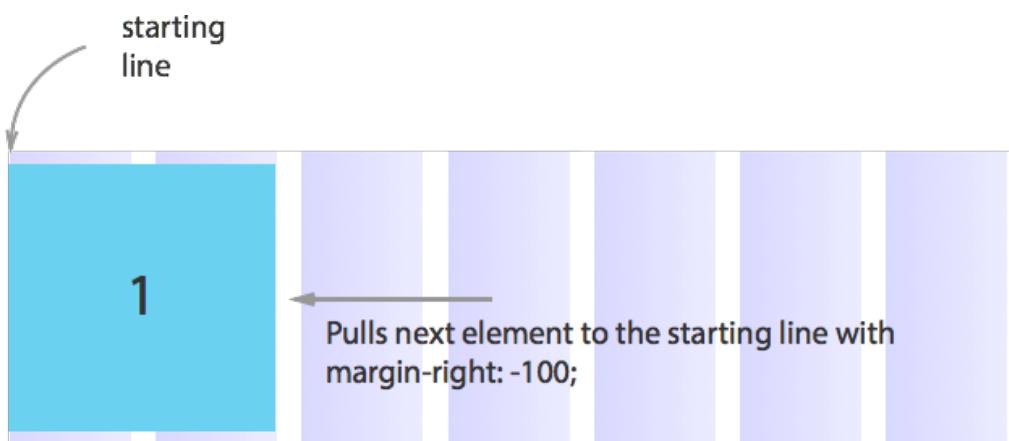
Say you need to create a 3-column layout.



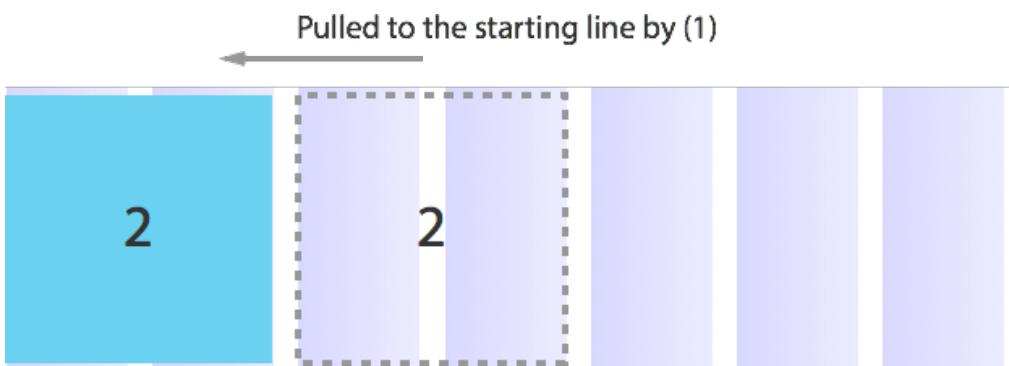
Each isolated element has a margin-left that positions it correctly, and a margin-right of -100% that draws the next element back to the left edge of the grid.

Let's look at one item at a time. item 1 is quite straightforward.

1. It has a margin-left of 0, so it sits on the start line.
2. Then it pulls the next item on the list back to the start line.

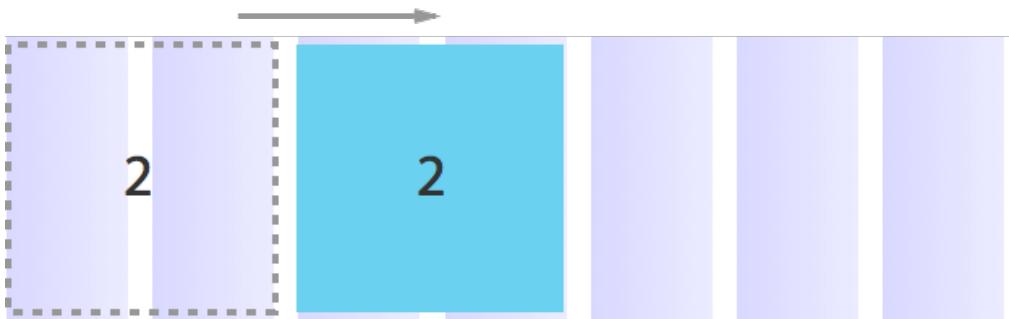


Item 2 has been pulled to the start line by item one before its own CSS properties kick in.

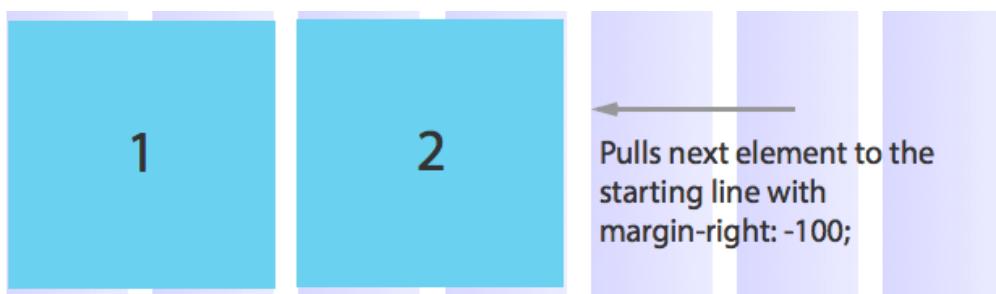


Item 2 is then able to position itself with the margin-left property, which makes it sit on the third column.

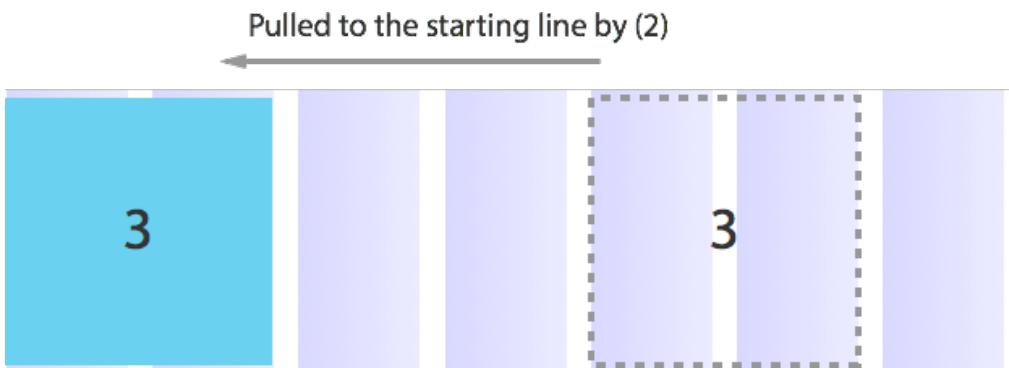
Pushed back into positioned by its own margin-left



Item 2 also has a `margin-right` property, which pulls the next item up to the start line.

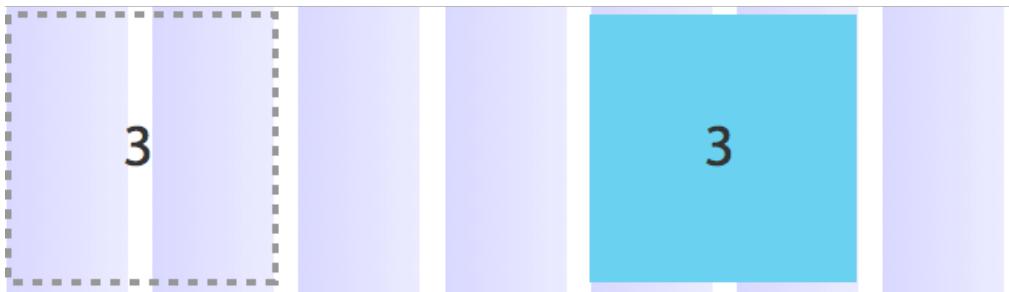


Item 3, like Item 2 before it, also gets pulled to the start line by Item 2.



It then uses the `margin-left` property to position itself on the 5th column.

Pushed back into position by its own margin-left



And it pulls the next item to the start line, and so on.

If you want any item to start on a new line, it has to clear the float from the previous item with the `clear: left` property.



But if it wants to start a new line and be positioned on the start line, it has to change its margin-left property back to 0 to place itself there.



And that's how isolate works :)

Equipped with this new knowledge, we can now move into how Susy uses the isolate technique. There are 3 ways for you to isolate elements with Susy.

1. Method 1: Set Global Output to Isolate
2. Method 2: Add Isolate argument to the Span Mixin
3. Method 3: Use the Isolate Mixin

Let's go through them one by one.

Method 1: Set Global Output to Isolate

The very first method to use Susy for isolation is to set the global `output` setting to `isolate` instead of `float`. This changes the `span` mixin to automatically accept the `$location` argument to isolate the element.

```
// Scss  
$susy : (  
  output: isolate,  
  // ...  
)
```

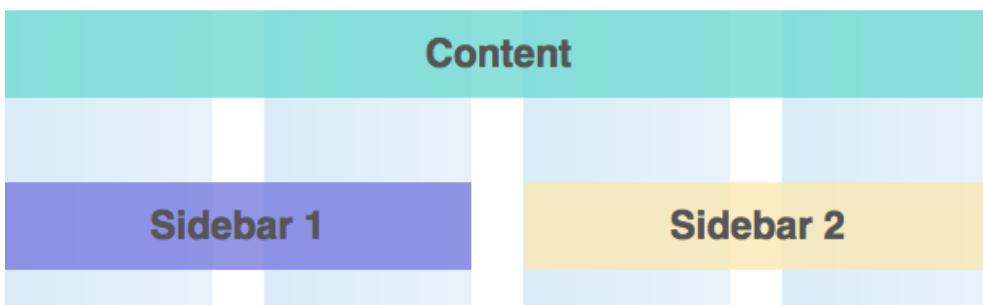
The layout changes from 4 columns to 12 columns at a certain breakpoint, say 960px. We also want to show the 12-column grid at the appropriate breakpoint.

```
// Scss
.wrap {
  @include container();

  @include susy-breakpoint(960px, 12) {
    @include show-grid();
  }
}
```

Let's take a look at the layout again.

Mobile view:

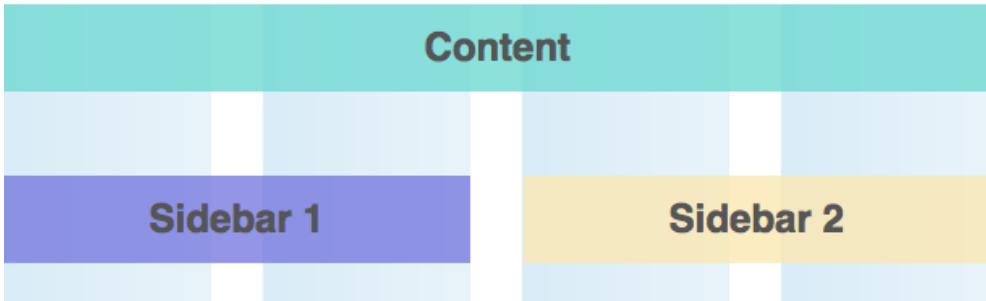


Desktop view:



`.content` takes up the full 100% width on mobile so we don't have to do anything for it. We still have to create the layout for `.sidebar1` and `.sidebar2` though. Each of them takes up 2 columns, with `.sidebar1` being the first item while `.sidebar2` being the last item.

```
.sidebar1 {  
  @include span(2 first);  
}  
  
.sidebar2 {  
  @include span(2 last);  
}
```



We have seen the `first` and `last` keyword multiple times in the previous chapters. These keywords are optional keywords that you can give to the `span()` and `gallery()` mixins. These keywords are the `$location` keyword.

The `$location` keyword is required when using the isolate technique with Susy. If no `$location` is found within the `span`, Susy will treat the `span` as a normal float element, even if the `output` is set to `isolate` in the `$susy` map.

```
// SCSS  
.sidebar1 {  
  @include span(2);  
}
```

```
/* CSS */
.sidebar1 {
  width: 47.36842%;
  float: left;
  margin-right: 5.26316%;
}
```

Because the `margin-left` and `margin-right` properties are integral in the isolate output, having any one element without a `margin-right` of `-100%` will cause the layout to collapse.

In this case, if we didn't add the `isolate` keyword to `.sidebar1`, `.sidebar2` would be thrown off to the right by the width of `sidebar1`.



Let's work on the desktop layout next.

The desktop layout



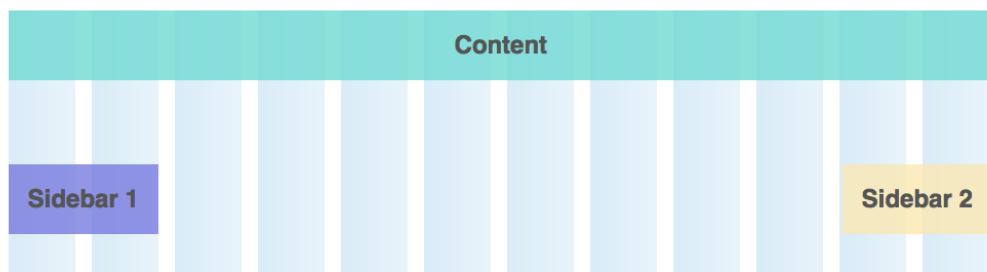
Both `.sidebar1` and `.sidebar2` takes up 2 of 12 columns on the desktop. `.sidebar1` is the first item while `.sidebar2` is the last item.

```
// SCSS
.sidebar1 {
  @include span(2 first);

  @include susy-breakpoint(960px, 12) {
    @include span(2 first);
  }
}

.sidebar2 {
  @include span(2 last);

  @include susy-breakpoint(960px, 12) {
    @include span(2 last);
  }
}
```



`.content` needs to take up 8 of 12 columns, and its starting position is on the 3rd column. This is where we have to use another form of the `$location` keyword.

We can use a number as the `$location` argument if we need to position an element at a specific column. This number must be preceded with an `at` keyword.

```
// SCSS
.content {
  @include susy-breakpoint(960px, 12) {
    @include span(8 at 3);
  }
}
```



[View Source code](#)

Method 2: Add the Isolate keyword to the Span Mixin

The second method you can use is to add the `isolate` keyword into the `span()` mixin. This works even if the global `output` is set to `float`.

This method is almost exactly the same as Method 1. The only difference is that you will have `isolate` arguments within each `span`.

```
.sidebar1 {  
  @include span(isolate 2 first);  
  
  @include susy-breakpoint(960px, 12) {  
    @include span(isolate 2 first);  
  }  
}  
  
.sidebar2 {  
  @include span(isolate 2 last);  
  
  @include susy-breakpoint(960px, 12) {  
    @include span(isolate 2 last);  
  }  
}  
  
.content {  
  @include susy-breakpoint(960px, 12) {  
    @include span(isolate 8 at 3);  
  }  
}
```

When you use the `isolate` keyword on the `span`, the local scope is set to `isolate` mode. This method allows you to use a float layout for most of your site and an isolate layout for a target portion.

If you find yourself writing `isolate` arguments all the time, it's probably a better idea to use the first method, which uses the `isolate` output instead.

Method 3: Use the Isolate Mixin

The third and final method to use the isolate technique is to use the `isolate()` mixin that Susy provides.

The `isolate()` mixin takes every argument and keyword that a `span()` mixin can take. However, it only outputs the `margin-left`, `margin-right` and `float` properties. It does not output the `width` property.

If you use this method, you must manually add a `width` to the element that you are trying to isolate. You can use the `span()` function for this.

Let's use `.sidebar1` on mobile view as an example. `.sidebar1` takes up 2 columns and is the first element. This translates to:

```
// Scss
.sidebar1 {
  width: span(2);
  @include isolate(first);
}
```

Notice the similarity between all three methods? The above will output the same thing as this:

```
// Scss
.sidebar1 {
  @include span(isolate 2 first);
}
```

You'd think the code for `.sidebar2` will look like this for the mobile view, right?

```
// Scss
.sidebar2 {
  width: span(2);
  @include isolate(last);
}
```

Nope! Unfortunately, the `isolate` mixin is not as powerful as the `span` mixin and cannot take in `last` arguments because there is no way for it to

calculate the width of the element and hence, its correct location. In this case, you will have to use the `at <number> style $location` argument instead.

And I'm sure you can breeze through the rest of the code yourself:

```
// Scss
.sidebar1 {
  width: span(2);
  @include isolate(first);
  @include susy-breakpoint(960px, 12) {
    width: span(2);
    @include isolate(first);
  }
}

.sidebar2 {
  width: span(2);
  @include isolate(3);
  @include susy-breakpoint(960px, 12) {
    width: span(2);
    @include isolate(11);
  }
}

.content {
  @include susy-breakpoint(960px, 12) {
    width: span(8);
    @include isolate(3);
  }
}
```

Another Row of Elements...

What we've discussed so far does not take into account additional rows of elements on the desktop view.

What if there is supposed to be another area for more content?

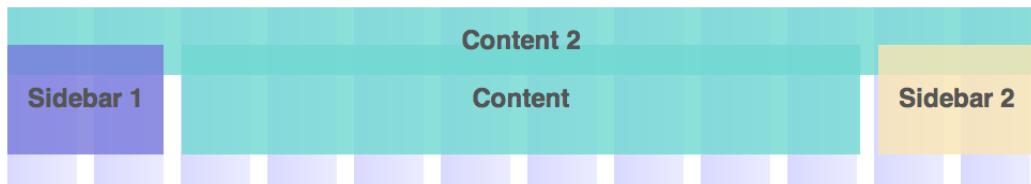


Let's get this to work as well.

First, we need to insert the extra HTML into the markup, below `.sidebar2`.

```
<!-- html -->
<div class="wrap">
    <!-- content and sidebar.. -->
    <div class="content-2">
        <h2>Content 2</h2>
    </div>
</div>
```

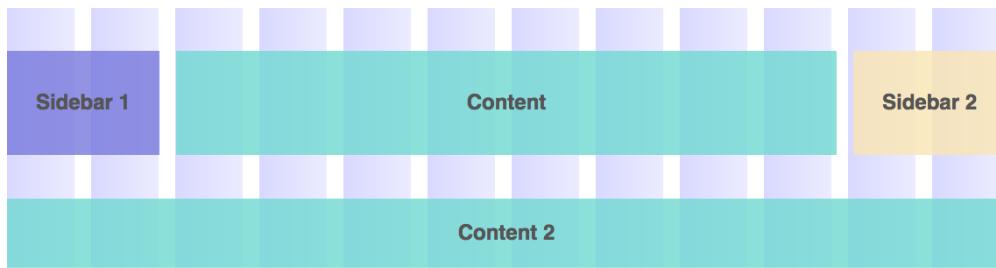
If you take a look at the HTML output now, you'll notice that it doesn't quite fit the way we want it to.



Think back to when we were learning about the `isolate` technique. We mentioned that you have to clear the previous floats if you wanted the elements to drop to the next level. This is precisely what we need to do here.

```
/* CSS */
.content-2 {
    clear: both;
}
```

And we're done!



A Quick Wrap Up

There are 3 methods to use the Isolate Technique with Susy. You can pick and choose which method you like the most and work with it.

The most important things about the Isolate Technique are these:

1. All `span()` mixins must contain the `$location` keyword
2. If `$location` refers to a specific column, use `at $location` instead. For example, if the element should be positioned at the 3rd column, use `at 3`
-

Now that we know how to isolate, let's begin another exciting chapter: Asymmetric Layouts.

Asymmetric Grids with Susy

The great thing about asymmetric layouts is that it gives you the freedom to design without the constraints of a same-column-width layout.

It's pretty rare to find great asymmetric grids out on the web because there are just so few of them around. Most grid frameworks out there simply don't cater for them. As of writing, those that do support asymmetric grids are Singularity GS, GridSetApp and Susy.

Most grid systems for the web box you in with a prescribed number of columns of the same width, meaning you have to fake the grid system you really need.

– Gridset App

One of the [best examples](#) I found about asymmetric grids is by [Nathan Ford](#). You can read more about how he designed his grid [here](#).

We are recreating this layout in two chapters.

In the first chapter, you will learn everything you need to set up asymmetric grids

You'll learn:

- How to write columns in an asymmetric grid
- How to write gutters in an asymmetric grid
- How to create an asymmetric grid container

In the second chapter, we will bring the concept further and make the asymmetric grid responsive.

You'll learn:

- How to get the \$context of an asymmetric grid
- How to work with nested asymmetric grids
- How to recreate the responsive asymmetric layout

There are 4 different breakpoints for this layout:

Small

This is Chapparal

Created by Adobe type designer Carol Twombly, Chaparral combines the legibility of slab serif designs popularized in the 19th century with the grace of 16th-century roman book lettering. By Adobe

The result is a versatile, hybrid slab-serif design, a unique addition to the Adobe Originals family of typefaces. Unlike “geometric” slab serif designs, Chaparral has varying letter proportions that give it an accessible and friendly appearance in all weights from light to bold.

ABOUT THE DESIGNER: CAROL TWOMBLY

American calligrapher and type designer, a graduate from Rhode Island School of Design where her professor was Charles Bigelow. Joined the digital typography program at Stanford University, also under Bigelow. Working from the Bigelow &

Medium

This is Chapparal

Created by Adobe type designer Carol Twombly, Chaparral combines the legibility of slab serif designs popularized in the 19th century with the grace of 16th-century roman book lettering. By Adobe

The result is a versatile, hybrid slab-serif design, a unique addition to the Adobe Originals family of typefaces. Unlike "geometric" slab serif designs, Chaparral has varying letter proportions that give it an accessible and friendly appearance in all weights from light to bold.

ABOUT THE DESIGNER: CAROL TWOMBLY

American calligrapher and type designer, a graduate from Rhode Island School of Design where her professor was Charles Bigelow. Joined the digital typography program at Stanford University, also under Bigelow. Working from the Bigelow & Holmes studio she designed Mirarae, which won her the 1984 Morisawa gold prize. Since 1988 she has been a staff designer

Large

This is Chapparal

Created by Adobe type designer Carol Twombly, Chaparral combines the legibility of slab serif designs popularized in the 19th century with the grace of 16th-century roman book lettering. By Adobe

The result is a versatile, hybrid slab-serif design, a unique addition to the Adobe Originals family of typefaces. Unlike "geometric" slab serif designs, Chaparral has varying letter proportions that give it an accessible and friendly appearance in all weights from light to bold.

Like the drought-resistant brush that blooms on the arid coastal range near Twombly's California home, Chaparral's highly functional design is surprisingly beautiful. Source: adobe.com

XLarge

This is Chapparal

Created by Adobe type designer Carol Twombly, Chaparral combines the legibility of slab serif designs popularized in the 19th century with the grace of 16th-century roman book lettering. By Adobe

The result is a versatile, hybrid slab-serif design, a unique addition to the Adobe Originals family of typefaces. Unlike "geometric" slab serif designs, Chaparral has varying letter proportions that give it an accessible and friendly appearance in all weights from light to bold.

Like the drought-resistant brush that blooms on the arid coastal range near Twombly's California home, Chaparral's highly functional design is surprisingly beautiful. Source: adobe.com

Let's begin by writing the CSS and HTML as usual.

Taking Care of CSS

Here's the usual CSS stuff :)

```
body {
  font-family: 'Chaparral Pro';
  color: rgba(0,0,0,0.75);
  line-height: 1.75;
  padding: 14.723624851955172% 0;
}

h1,h2,h3,h4,h5,h6 {
  font-weight: 400;
  margin: 0;
}

h1 {
  font-size: 7.5vw;
  font-style: italic;
  line-height: 1;
  margin-bottom: 1ex;
}

h2 {
  font-size: 3ex;
  margin-bottom: 2ex;
}

h3 {
  text-transform: uppercase;
  letter-spacing: 0.1em;
  padding: 8ex 0 0;
}
```

Remember to place the clearfix mixin at the top of the Sass file!

```

@mixin cf {
  &:after {
    content: "";
    display: table;
    clear: both;
  }
}

```

Writing the HTML

The HTML is split into 3 sections, each section with its own section title. In addition, we need to add a `.wrap` to contain the sections within. A picture speaks more than a thousand words here.

Wrap

This is Chapparal

Section

Created by Adobe type designer Carol Twombly, Chapparal combines the legibility of slab serif designs popularized in the 19th century with the grace of 16th-century roman book lettering. By Adobe

The result is a versatile, hybrid slab-serif design, a unique addition to the Adobe Originals family of typefaces. Unlike "geometric" slab serif designs, Chapparal has varying letter proportions that give it an accessible and friendly appearance in all weights from light to bold.

Like the drought-resistant brush that blooms on the arid coastal range near Twombly's California home, Chapparal's highly functional design is surprisingly beautiful. Source: adobe.com

Section

ABOUT THE DESIGNER: CAROL TWOMBLY

American calligrapher and type designer, a graduate from Rhode Island School of Design where her professor was Charles Bigelow. Joined the digital typography program at Stanford University, also under Bigelow. Working from the Bigelow & Holmes studio she designed Mirarae, which won her the 1984 Morisawa gold prize. Since 1988 she has been a staff designer at Adobe.

During the 1994 ATypI conference in San Francisco, she was awarded the prestigious Prix Charles Peignot, given (occasionally) to outstanding type designers under the age of 35. Source: myfonts.com

Section

YET ANOTHER SECTION

In its natural regime, chaparral is characterized by infrequent fires, with intervals ranging between 10–15 years to over a hundred years. Mature chaparral (stands that have been allowed greater intervals between fires) is characterized by nearly impenetrable, dense thickets (except the more open chaparral of the desert). These plants are highly flammable. They grow as woody shrubs with hard and small leaves; are non-leaf dropping (non-deciduous); and are drought tolerant. After the first rains following a fire, the landscape is dominated by soft-leaved non-woody annual plants, known as fire followers, which die back with the summer dry period.

Similar plant communities are found in the four other Mediterranean climate regions around the world, including the Mediterranean Basin (where it is known as maquis), central Chile (where it is called matorral), South African Cape Region (known there as fynbos), and in Western and Southern Australia (as kwongan). According to the California Academy of Sciences, Mediterranean shrubland contains more than 20% of the world's plant diversity. The word chaparral is a loan word from Spanish chapparo, meaning both "small" and "dwarf" evergreen oak, which itself comes from the Basque word txapar, with exactly the same meaning.

Conservation International and other conservation organizations consider the chaparral to be a biodiversity hotspot – a biological community with a large number of different species – that are under threat by human activity. Source: Wikipedia

Each of these sections will be given the `.content` class. Hence the HTML looks like:

```
<div class="wrap">
  <div class="content"> <!-- content within --> </div>
  <div class="content"> <!-- content within --> </div>
  <div class="content"> <!-- content within --> </div>
</div>
```

We can also see that the first section is slightly different from the rest, so let's give it an `.intro` class.

```
<div class="wrap">
  <div class="content intro"> <!-- content within --> </div>
  <div class="content"> <!-- content within --> </div>
  <div class="content"> <!-- content within --> </div>
</div>
```

Each of the sections contain a section header and a few paragraphs of text that eventually splits into two columns. If the text columns are of equal spacing, we could have used CSS columns to output them.

In this case, however, we have to manually split the text into two (left and right) sections.

```
<div class="wrap">
  <div class="content intro">
    <h2> <!-- Section Header --> </h2>
    <div class="content__left"> <!-- paragraphs of text within --></div>
      <div class="content__right"> <!-- paragraphs of text within --></div>
    </div>
    <div class="content">
      <h3> <!-- Section Header --> </h3>
      <div class="content__left"> <!-- paragraphs of text within --></div>
        <div class="content__right"> <!-- paragraphs of text within --></div>
      </div>
      <div class="content">
        <h3> <!-- Section Header --> </h3>
        <div class="content__left"> <!-- paragraphs of text within --></div>
          <div class="content__right"> <!-- paragraphs of text within --></div>
        </div>
      </div>
    </div>
  </div>
```

The completed HTML with content is:

```
<div class="wrap">
  <div class="content intro">
    <h1 class="title">This is Chapparal</h1>
    <h2 class="subtitle">
      Created by Adobe type designer Carol Twombly, Chaparral
      combines the legibility of slab serif designs popularized in the
      19th century with the grace of 16th-century roman book
      lettering. By Adobe
    </h2>

    <div class="content__left">
      <p>
        The result is a versatile, hybrid slab-serif design, a
```

unique addition **to** the Adobe Originals family of typefaces. Unlike “geometric” slab serif designs, Chaparral has varying letter proportions that give it an accessible and friendly appearance in all weights **from** light **to** bold.

```
</p>
</div>
```

```
<div class="content__right">
    <p>Like the drought-resistant brush that blooms on the arid coastal range near Twombly’s California home, Chaparral’s highly functional design is surprisingly beautiful. Source: adobe.com</p>
```

```
</div>
</div>
```

```
<div class="content">
    <h3>About the designer: Carol Twombly</h3>
    <div class="content__left">
        <p>
            American calligrapher and type designer, a graduate from Rhode Island School of Design where her professor was Charles Bigelow. Joined the digital typography program at Stanford University, also under Bigelow. Working from the Bigelow & Holmes studio she designed Mirarae, which won her the 1984 Morisawa gold prize. Since 1988 she has been a staff designer at Adobe.
```

```
</p>
</div>
```

```
<div class="content__right">
```

```
<p>
```

During the **1994** ATypI conference in San Francisco, she was awarded the prestigious Prix Charles Peignot, given (occasionally) **to** outstanding type designers under the age of **35**. Source: myfonts.com

```
</p>
</div>
```

```
</div>
</div>
```

```
<div class="wrap">
  <div class="content">

    <h3>Yet another section </h3>

    <div class="content__left">
      <p>
        In its natural regime, chaparral is characterized by infrequent fires, with intervals ranging between 10–15 years to over a hundred years. Mature chaparral (stands that have been allowed greater intervals between fires) is characterized by nearly impenetrable, dense thickets (except the more open chaparral of the desert). These plants are highly flammable. They grow as woody shrubs with hard and small leaves; are non-leaf dropping (non-deciduous); and are drought tolerant. After the first rains following a fire, the landscape is dominated by soft-leaved non-woody annual plants, known as fire followers, which die back with the summer dry period.
      </p>
    </div>
    <div class="content__right">
      <p>
        Similar plant communities are found in the four other Mediterranean climate regions around the world, including the Mediterranean Basin (where it is known as maquis), central Chile (where it is called matorral), South African Cape Region (known there as fynbos), and in Western and Southern Australia (as kwongan). According to the California Academy of Sciences, Mediterranean shrubland contains more than 20% of the world's plant diversity. The word chaparral is a loan word from Spanish chaparro, meaning both "small" and "dwarf" evergreen oak, which itself comes from the Basque word txapar, with exactly the same meaning.
      </p>
      <p>
        Conservation International and other conservation organizations consider the chaparral to be a biodiversity hotspot – a biological community with a large number of
      </p>
    </div>
  </div>
</div>
```

```
different species – that are under threat by human activity.  
Source: Wikipedia  
      </p>  
      </div>  
    </div>  
</div>
```

The Susy Map

Asymmetric grids rely on the isolate technique to position elements correctly, so we have to set `output` to `isolate` in the `$susy` map to build asymmetric grids.

```
$susy: (  
  output: isolate,  
  global-box-sizing: border-box,  
  debug: (image: show)  
);  
  
@include border-box-sizing;
```

The initial setup is now complete.

Next, we need to find out how to tell Susy that we're using an asymmetric grid.

The Asymmetric Map

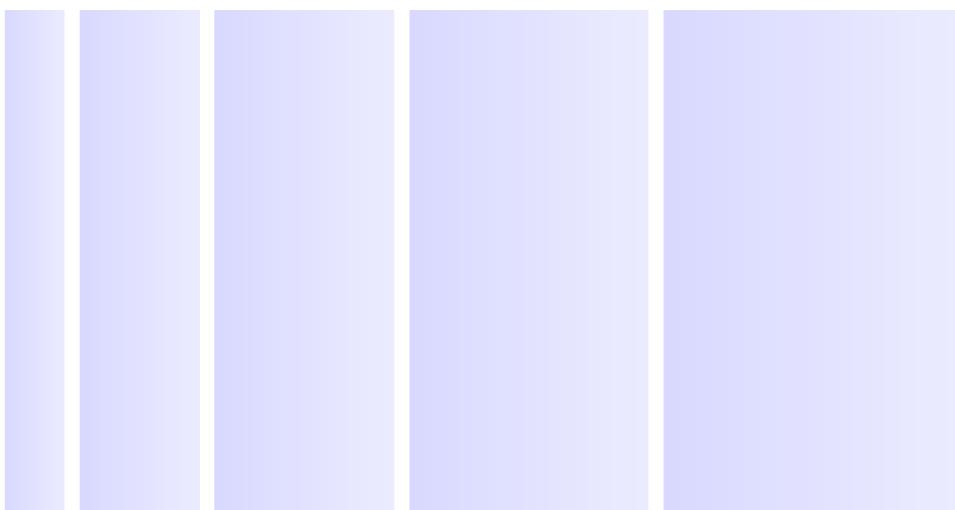
Columns settings

The `columns` setting within the `$susy` map determines whether we are working on an asymmetric grid. Susy will create an asymmetric grid if a Sass list is given to the `columns` setting.

A Sass list is an array of items separated by spaces or commas:

```
$sass-list: 1 2 3 4 5;  
$sass-list-comma: 1, 2, 3, 4, 5;
```

Here, we have a list of 5 items, 1, 2, 3, 4 and 5. We will get an asymmetric grid with 5 columns if we placed this list into the columns setting. You can use either of the formats mentioned (I prefer to stick with the space-delimited version).



It is not necessary for the numbers in the list to be complete integers. Decimal points are perfectly acceptable:

```
$susy : (  
  columns: 0.25 0.5 0.75 1 1.25  
)
```

However, they cannot contain any units.

```
$susy: (  
  columns: 10px 20px 30px // This is not allowed  
)
```

A very important point for setting asymmetric grid columns is that the base number **must be 1**. The other column values in the list should be a relative multiple of the base column `1`.

```
$susy: (
  columns: 1 2 3 4 5; // This is allowed
)
$susy: (
  columns: 2 3 4 5 6 // This is not allowed
);
```

Say you have an asymmetric grid with the following column sizes:

```
$susy: (
  columns: 20px 30px 40px 50px // This is not allowed
);
```

That is not allowed, so we have to convert the columns into a list, where there is a common base of `1`. If we use 20px as the base, we will divide each column by 20 pixels. That would give us:

```
$susy: (
  columns: 1 1.5 2 2.5 // This is allowed and can be used
)
```

The example below sums up everything within this section:

```
columns: 20px 50px 10px // This is not allowed
columns: 2 3 4 5 6 // This is not allowed.

columns: 1 2 3 4 5 // This is okay, because it has the number 1
columns: 0.5 1 1.5 2 2.5 // This is okay as well
```

Gutter settings

The gutter settings within an asymmetric grid will be calculated based on the **base multiple of the asymmetric grid (in other words, 1)**.

In the example used above, the base multiple of 1 has a value of 20px. If we have a gutter size of 10px, gutter is 10px divided by 20px.

```
// Scss
$susy : (
  columns: 1 1.5 2 2.5
  gutters: 10 / 20; // 10px ÷ 20px
)
```

Setting Up The Asymmetric Grid

We know how to tell Susy that we're working on an asymmetric grid by using a Sass list in the `columns` setting. Let's apply this knowledge and create the grid for this layout.

Have a look at the smallest view again:

Small Layout

This is Chaparral

Created by Adobe type designer Carol Twombly, Chaparral combines the legibility of slab serif designs popularized in the 19th century with the grace of 16th-century roman book lettering. By Adobe

The result is a versatile, hybrid slab-serif design, a unique addition to the Adobe Originals family of typefaces. Unlike “geometric” slab serif designs, Chaparral has varying letter proportions that give it an accessible and friendly appearance in all weights from light to bold.

ABOUT THE DESIGNER: CAROL

TWOMBLY

American calligrapher and type designer, a graduate from Rhode Island School of Design where her professor was Charles Bigelow. Joined the digital typography program at Stanford University, also under Bigelow. Working from the Bigelow &

We have to measure the width of each column and the width of the gutters if we were to recreate this asymmetric grid:



- The first column is 42px wide
- The second column is 288px wide.
- The gutter is 10px.

It would be sensible to use the first column as the base multiple of 1 because it is the smallest column. In this case, we will have the following values:

- First column: 1 ($42 \div 42$)
- Second column: 6.857 ($288 \div 42$).
- Gutters: $10 \div 42$

One other quirk in this small layout is that it's not completely fluid. It has a max-width. We want Susy to calculate the max-width for us in pixels so we need to set `column-width` to 42px, since it is used as the base (more on `column-width` in a later chapter).

```
// Scss
$small : (
  columns: 1 6.857142857,
  gutter: 10 / 42,
  column-width: 42px
)
```

Medium Layout



We have 4 columns in the medium layout:

- First column is 54px
- Second column is 88px
- Third column is 370px
- Fourth column is 54px
- Gutter is 12px

Using a similar approach as the small layout, we'll divide everything by 54. This will yield the following values:

- First column: 1 ($54 \div 54$)
- Second column: 1.62962963 ($88 \div 54$)
- Third column: 6.851851852 ($370 \div 54$)
- Fourth column: 1 ($54 \div 54$)
- Gutter: 12 / 54

```
// Scss
$medium : (
  columns: 1 1.62962963 6.851851852 1,
  gutter: 12 / 54,
);
```

Large Layout



The large layout and extra large layout are so similar to each other that we can create them together. The extra large layout is slightly more complex and that's what we're basing this map on.

Essentially, we have 5 columns:

- First column is 103px
- Second column is 435px
- Third column is 257px
- Fourth column is 416px
- Fifth column is 269px
- Gutter is 32px

We can choose 103px to be the base

- First column: 1 ($103 \div 103$)
- Second column: 4.223300971 ($435 \div 103$)
- Third column: 2.495145631 ($257 \div 103$)
- Fourth column: 4.038834951 ($416 \div 103$)
- Fifth column: 2.553398058 ($269 \div 103$)
- Gutter: 32 / 103

```
$large: (
  columns: 4.223300971 1 2.495145631 4.038834951 2.553398058,
  gutter: 32 / 103,
);
```

Note: We are super precise with the decimal points in this example because We're trying to replicate the layout as best as we can. You don't need to be as precise if you already know the proportions for your layout.

While we are making the map, let's also include variables for the breakpoints of this layout.

```
$bp-med: 530px;  
$bp-large: 700px;  
$bp-xlarge: 1600px;
```

Creating The Susy Container

We created `$small`, `$medium` and `$large` maps to hold the layouts' different breakpoints instead of adding them to the `$susy` map this time round.

In order to create the Susy container with the smallest map, we have to use the `layout()` mixin to substitute the default `$susy` map with the one that we want.

```
// Scss  
@include layout($small);
```

This `layout()` mixin overwrites the `$susy` map settings with any setting that we declared in the `$small` map.

Now, we can create the container.

```
// Scss  
.wrap {  
  @include container;  
}
```

This time, we didn't include a `max-width` argument within the container because we want Susy to calculate the `max-width` for us. From the `$bp-med` breakpoint onwards, however, we want the max-width to become fluid at 100%.

We can force this by setting a `max-width` property. We also want to show the grid for debugging purposes.

```
.wrap {  
  @include container;  
  @include susy-breakpoint($bp-med, $med) {  
    max-width: 100%;  
    @include show-grid;  
  }  
  
  @include susy-breakpoint($bp-large, $large) {  
    @include show-grid;  
  }  
}
```

[View Source code](#)

A Quick Wrap Up

We went through how to write the HTML and CSS for this asymmetric grid (which is rather complex), and figured out how to tell Susy that we are making an asymmetric grid.

We went ahead and created layout maps for the breakpoints at each layout that we will soon apply to create our grid.

The next step is the fun part where you'll learn how to create the grid with what we've made here!

Asymmetric Grids With Susy (Part 2)

We managed to create the asymmetric grid maps in the previous chapter. Here's the fun part on applying them and creating the asymmetric grid we have been talking about.

In this chapter, you'll learn:

- How to get the `$context` of an asymmetric grid
- How to work with nested asymmetric grids
- How to recreate the responsive asymmetric layout

Laying It Out With Susy

Let's begin with a quick recap of the maps that we created in the last chapter:

```
// Scss
$small: (
  columns: 1 6.85714,
  gutter: 10 / 42,
  column-width: 42px
);

$med: (
  columns: 1 1.62963 6.85185 1,
  gutter: 12 / 54
);

$large: (
  columns: 4.2233 1 2.49515 4.03883 2.5534,
  gutter: 32 / 103
);
```

How are we supposed to write the grid if we had to work with such complex numbers?!

Thankfully, we can ignore the strings of decimal points when creating the asymmetric grid. All we need to know is the number of columns in each layout.

- `$small` has 2 columns
- `$medium` has 4 columns and
- `$$large` has 5 columns.

Once we know the number of columns in each layout, we know the context that we're working with. The `$context` is 2 on the `$small` layout, 4 on the `$medium` layout and 5 on the `$large` layout.

With this knowledge, we can work on the small layout:

Small Layout

If we take a look at the small layout once again:

This is Chapparal

Created by Adobe type designer Carol Twombly, Chaparral combines the legibility of slab serif designs popularized in the 19th century with the grace of 16th-century roman book lettering. By Adobe

The result is a versatile, hybrid slab-serif design, a unique addition to the Adobe Originals family of typefaces. Unlike “geometric” slab serif designs, Chaparral has varying letter proportions that give it an accessible and friendly appearance in all weights from light to bold.

ABOUT THE DESIGNER: CAROL

TWOMBLY

American calligrapher and type designer, a graduate from Rhode Island School of Design where her professor was Charles Bigelow. Joined the digital typography program at Stanford University, also under Bigelow. Working from the Bigelow &

You can see that the content within `.intro` takes up a full two columns while the content within the second section only takes up the last column.

Since we know the content consists of both `.content__left` and `.content__right`, we can use the `span()` mixin to create the layout.

We also need to give both `.content__left` and `.content__right` a `clear: both` property to make sure they do not overlap each other.

```
// Scss
.content__left,
.content__right {
  clear: both;
  @include span(1 last);
}
```

Since the content on the intro takes up the two columns, we can create a selector with a higher specificity to override what we already declared.

```
// Scss
.intro {
  .content__left,
  .content__right {
    @include span(full);
  }
}
```

Remember to give `.content` a clearfix since all child elements within it are floated.

```
// Scss
.content {
  @include cf;
}
```

This is Chapparal

Created by Adobe type designer Carol Twombly, Chaparral combines the legibility of slab serif designs popularized in the 19th century with the grace of 16th-century roman book lettering. By Adobe

The result is a versatile, hybrid slab-serif design, a unique addition to the Adobe Originals family of typefaces. Unlike “geometric” slab serif designs, Chaparral has varying letter proportions that give it an accessible and friendly appearance in all weights from light to bold.

ABOUT THE DESIGNER: CAROL TWOMBLY

American calligrapher and type designer, a graduate from Rhode Island School of Design where her professor was Charles Bigelow. Joined the digital typography program at Stanford University, also under Bigelow. Working from the Bigelow &

[View Source code](#)

And we built the small layout already! Let's move on to the medium layout next.

Medium Layout

Let's have a look at the medium layout again:

This is Chapparal

Created by Adobe type designer Carol Twombly, Chaparral combines the legibility of slab serif designs popularized in the 19th century with the grace of 16th-century roman book lettering. By Adobe

The result is a versatile, hybrid slab-serif design, a unique addition to the Adobe Originals family of typefaces. Unlike “geometric” slab serif designs, Chaparral has varying letter proportions that give it an accessible and friendly appearance in all weights from light to bold.

ABOUT THE DESIGNER: CAROL TWOMBLY

American calligrapher and type designer, a graduate from Rhode Island School of Design where her professor was Charles Bigelow. Joined the digital typography program at Stanford University, also under Bigelow. Working from the Bigelow & Holmes studio she designed Mirarae, which won her the 1984 Morisawa gold prize. Since 1988 she has been a staff designer

We can see there are now 4 columns in the medium layout. We have to change the background grid to match these four columns. We also want to have a max-width of 100% instead of asking Susy to calculate it for us automatically.

```
.wrap {  
  @include container;  
  @include susy-breakpoint($bp-med, $med) {  
    max-width: 100%;  
    @include show-grid;  
  }  
}
```

Writing the asymmetric layout is exactly the same as writing the symmetric layout once you know the number of columns and the context used.

In this case, we can see that the content of the layout spans the two middle columns and it is located on the second column.

```
// SCSS
.content {
  @include cf;
  @include susy-breakpoint(530px, $med) {
    @include span(2 at 2); // 2 columns, beginning on the second
    column
  }
}
```

The result is a versatile, hybrid slab-serif design, a unique addition to the Adobe Originals family of typefaces. Unlike “geometric” slab serif designs, Chaparral has varying letter proportions that give it an accessible and friendly appearance in all weights from light to bold.

Like the drought-resistant brush that blooms on the arid coastal range near Twombly’s California home, Chaparral’s highly functional design is surprisingly beautiful.

Source: [adobe.com](#)

ABOUT THE DESIGNER: CAROL TWOMBLY

American calligrapher and type designer, a graduate from Rhode Island School of Design where her professor

... Charles Bigelow Trained the digital

We can see that both `.content__left` and `.content__right` columns are off slightly from the grid because they still use styles from the small layout.

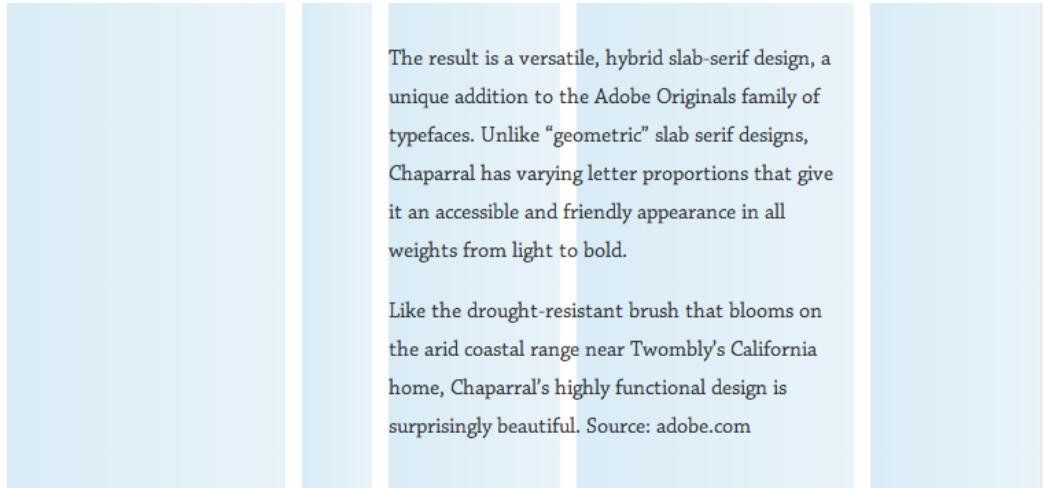
We need to find out how to obtain the correct context for `.content__left` and `.content__right` to make sure we can place them properly.

First of all, we know that the context for the medium layout is

(1 1.62963 6.85185 1) . The context for the .content is the middle two columns, which is (1.62963 6.85185) .

We also know that both .content_left and .content_right take up the right column in this context. One way we could write the span() mixin is:

```
// Scss
.content_left,
.content_right {
  clear: both;
  @include span(1 last);
  @include breakpoint($bp-med) {
    @include span(1 of (1.62963 6.85185) last);
  }
}
```



The result is a versatile, hybrid slab-serif design, a unique addition to the Adobe Originals family of typefaces. Unlike “geometric” slab serif designs, Chaparral has varying letter proportions that give it an accessible and friendly appearance in all weights from light to bold.

Like the drought-resistant brush that blooms on the arid coastal range near Twombly’s California home, Chaparral’s highly functional design is surprisingly beautiful. Source: adobe.com

It's confusing and incredibly difficult to maintain if we have to keep writing the columns like this :(

Thankfully, there is a way to find the correct nested context without resorting to copying the columns setting from the map.

Finding The Nested Context

First, we need a way to get the columns of `$med`. Since `$med` is written as a Sass map, we can use the `map-get()` function within Sass to get the value of the `columns` key.

The `map-get()` function takes in two arguments. The first argument is the map to extract the value from and the second argument is the key we are targeting.

```
// Scss  
$value: map-get($map, key);
```

We are looking at the `$med` map and the `columns` key. So we write:

```
// Scss  
$columns: map-get($med, columns);
```

We can use `@debug` to log out the value of `$columns` to see if we have the correct value. When compiling, Sass will log the `@debug` values within the terminal or application that you are running it with.

```
// Scss  
@debug $columns; // returns 1 1.62963 6.85185 1
```

Great! Now, we have the `$med` `columns`.

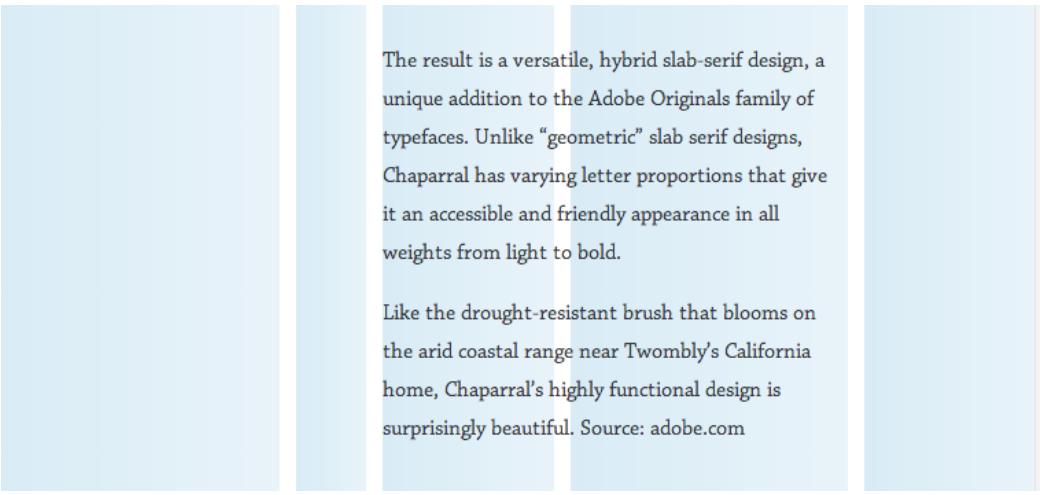
Next, we need to find the context. We can get this context with the `nested()` function that Susy provides. We know that `.content` is written with `span(2 of $columns at 2)`. We can get the context by replacing the `span()` mixin with the `nested()` function.

```
$context: nested( 2 of $columns at 2);  
@debug $context; // returns 1.62963 6.85185
```

And here's the context we're looking for! We can use this context within the `span()` mixin.

```
.content__left,  
.content__right {  
  clear: both;  
  @include span(1 last);  
  
  @include breakpoint($bp-med) {  
    $columns: map-get($med, columns);  
    $context: nested(2 of $columns at 2);  
    @include span(1 of $context last);  
  }  
}
```

And we get the same output.



The result is a versatile, hybrid slab-serif design, a unique addition to the Adobe Originals family of typefaces. Unlike “geometric” slab serif designs, Chaparral has varying letter proportions that give it an accessible and friendly appearance in all weights from light to bold.

Like the drought-resistant brush that blooms on the arid coastal range near Twombly's California home, Chaparral's highly functional design is surprisingly beautiful. Source: adobe.com

Although we managed to get the context working, this style of writing isn't ideal because we introduce global variables into our Sass code, and there's unnecessary repetition, making our code less DRY.

We can shorten things up by combining some of the code. For instance,

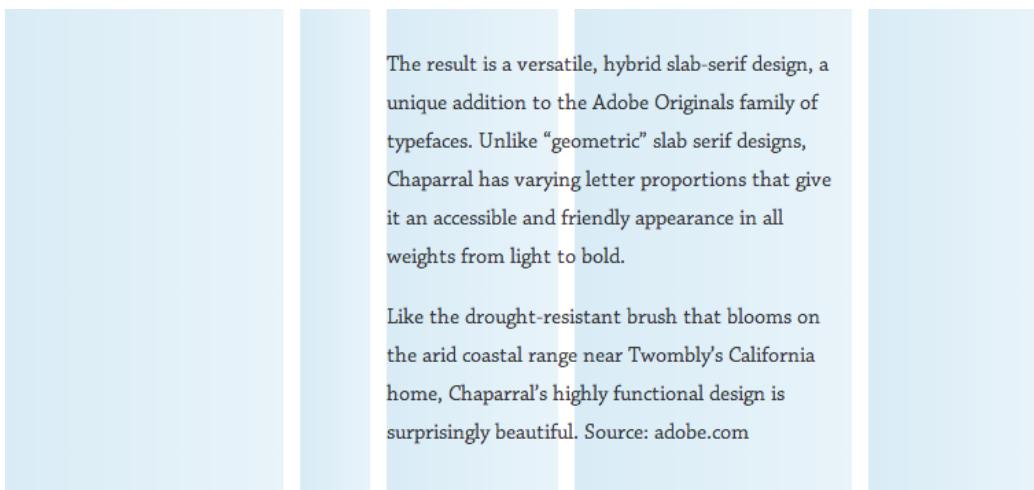
`$context` can be combined with `$columns` to become:

```
$context: nested(2 of map-get($med, columns) at 2);
```

We can also omit the `$context` variable if we used the `nested()` mixin directly.

```
.content__left,  
.content__right {  
  clear: both;  
  @include span(1 last);  
  
  @include breakpoint($bp-med) {  
    @include nested(2 of map-get($med, columns) at 2) {  
      @include span(1 last);  
    }  
  }  
}
```

And we get the same layout as well!



Note: The `susy-breakpoint()` mixin won't work when you're trying to get the context at the same time. This is why we used `breakpoint()` and `nested()` instead.

There's one more thing we have to address in the medium layout. Notice how the content intro paragraph now appears the same as the rest of the paragraphs? We need to fix that.

Since the intro paragraph is only different on the smallest breakpoint, we can use a `max-width` media query to make sure it only targets the smallest layout.

```
// Scss
.intro {
  .content__left, .content__right {
    @include breakpoint(max-width $bp-med - 1) {
      @include span(full);
    }
  }
}
```

We're done with the medium layout now. Let's replicate this process for the large layout.

[View Source code](#)

Large Layout

Just as we viewed the medium layout before working on it, let's first have a look at the large layout again.

This is Chapparal

Created by Adobe type designer Carol Twombly, Chaparral combines the legibility of slab serif designs popularized in the 19th century with the grace of 16th-century roman book lettering. By Adobe

The result is a versatile, hybrid slab-serif design, a unique addition to the Adobe Originals family of typefaces. Unlike “geometric” slab serif designs, Chaparral has varying letter proportions that give it an accessible and friendly appearance in all weights from light to bold.

Like the drought-resistant brush that blooms on the arid coastal range near Twombly’s California home, Chaparral’s highly functional design is surprisingly beautiful. Source: adobe.com

The first thing we notice is that the background grid has changed into 5 columns from the `$large` map that we made previously. The next step is to update the background grid so we can make sure we’re on the right track!

```
.wrap {  
  // ...  
  @include susy-breakpoint($bp-large, $large) {  
    @include show-grid;  
  }  
}
```

We can also see that `.content` now spans 3 columns in the `$large` layout and it starts on the second column.

```
// SCSS
.content {
    // ...
    @include susy-breakpoint($bp-large, $large) {
        @include span(3 at 2);
    }
}
```

With this small change, we have the background grid in place and `.content` now spans the full 3 columns. The only thing that's short right now is that `.content__left` and `.content__right` are off the grid.

The result is a versatile, hybrid slab-serif design, a unique addition to the Adobe Originals family of typefaces. Unlike "geometric" slab serif designs, Chaparral has varying letter proportions that give it an accessible and friendly appearance in all weights from light to bold.

Like the drought-resistant brush that blooms on the arid coastal range near Twombly's California home, Chaparral's highly functional design is surprisingly beautiful.

Source: [adobe.com](#)

Group 1

ABOUT THE DESIGNER: CAROL TWOMBLY

American calligrapher and type designer, a graduate from Rhode Island School of Design where her professor

was Charles Bigelow. Trained the digital

Group 2

The good news is we already know what's wrong and we know how to fix it! We have to use the same process as when we figured out the context for the medium layout.

First, we need to get the `columns` key from the `$large` map with `map-get($large, columns)`.

Next, we need to get the context with the `nested()` mixin passing in the columns, context and location that were used,

`nested(3 of map-get($large, columns) at 2)`, in other words.

```
.content__left,  
.content__right {  
  // ...  
  @include breakpoint($bp-large) {  
    @include nested(3 of map-get($large, columns) at 2) {  
      @include span(2 last);  
    }  
  }  
}
```

And we can see the magic happen without having to calculate anything!

The result is a versatile, hybrid slab-serif design, a unique addition to the Adobe Originals family of typefaces. Unlike "geometric" slab serif designs, Chaparral has varying letter proportions that give it an accessible and friendly appearance in all weights from light to bold.

Group 3

Like the drought-resistant brush that blooms on the arid coastal range near Twombly's California home, Chaparral's highly functional design is surprisingly beautiful.

Source: [adobe.com](#)

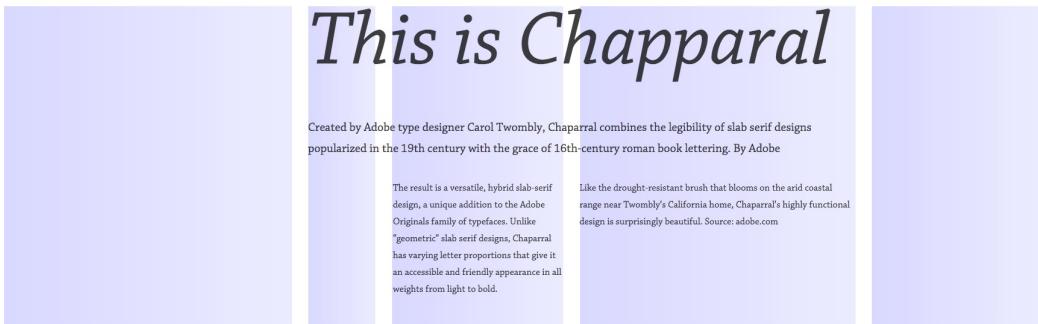
Group 4

[View Source code](#)

We have some final tweaks to make to the extra large breakpoint at 1600px. Let's do that now.

Xlarge Layout

Just like what we've done before, let's first have a look at the xlARGE layout.



We can see that the background grid remains the same as the background grid from the `$large` layout. This means we don't have to fiddle around with the `.wrap` and background grid.

The only thing that has changed is that `.content__left` now only takes up 1 column and `.content__right` only takes up 1 column as well.

Since both `.content__left` and `.content__right` are nested grid items, we have to use the same method to find their context. The context we need is `nested(3 of map-get($large, columns) at 2)` since the layout didn't change from the large breakpoint.

```
.content__left {  
  @include breakpoint($bp-xlarge) {  
    @include nested(3 of map-get($large, columns) at 2) {  
      @include span(1 at 2);  
    }  
  }  
}  
  
.content__right {  
  @include breakpoint($bp-xlarge) {  
    @include nested(3 of map-get($large, columns) at 2) {  
      @include span(1 last);  
    }  
  }  
}
```

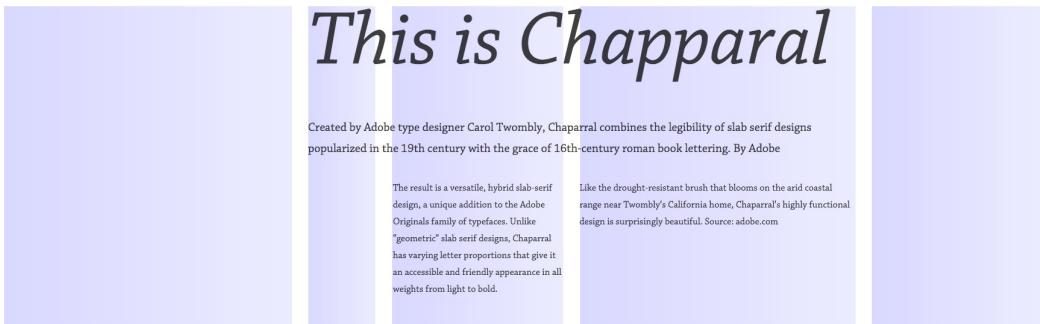
The result is a versatile, hybrid slab-serif design, a unique addition to the Adobe Originals family of typefaces. Unlike "geometric" slab serif designs, THOM has varying letter proportions that give it an accessible and friendly appearance in all weights. American calligrapher and type designer, a graduate from Rhode Island School of Design where her professor was Charles Holmes, she designed THOM while working at the University of California, Berkeley. In its natural regime, chaparral is characterized by infrequent fires, with intervals ranging between 10–15 years to over a hundred years. Mature chaparral stands that have been allowed greater intervals between fires) is characterized by

We're on track with the width and columns now, there's only one more thing to be done.

Both `.content__left` and `.content__right` have a `clear: both` property built into them. This `clear: both` is causing `.content__right` to create a new row for itself. Since we want both of them to be on the same row, we need to remove this `clear: both` property from `.content__right`.

```
.content__right {  
  @include breakpoint($bp-xlarge) {  
    clear: none;  
    @include nested(3 of map-get($large, columns) at 2) {  
      @include span(1 last);  
    }  
  }  
}
```

And we get the layout we want.



You could flatten and make the code slightly DRYer by combining the `.content__left` and `.content__right` selectors in the breakpoint. This way, we write 1 less context.

```
@include breakpoint($bp-xlarge) {  
  @include nested(3 of map-get($large, columns) at 2) {  
    .content__left {  
      @include span(1 at 2);  
    }  
    .content__right {  
      clear: none;  
      @include span(1 last);  
    }  
  }  
}
```

That's how you make an asymmetric grid!

[View Source code](#)

A Quick Wrap Up

The asymmetric grid we just made is way more complex than what you will normally use. The key challenge behind asymmetric grids are to:

1. Create the correct `$grid-map` at the different breakpoints
2. Get the correct context

There's nothing stopping you from working your magic around anything in Susy if you have understood the whole process of getting this grid.

We have always looked at creating fluid grids so far in the book. Let's take a look at how to use Susy to create static-width grids next.

Static Grids With Susy

Sometimes you run into situations where a fluid layout won't fit your needs. It might be because you feel that layouts should be in pixels, or you feel that typography measures becomes too long at many viewport sizes to make reading comfortable.

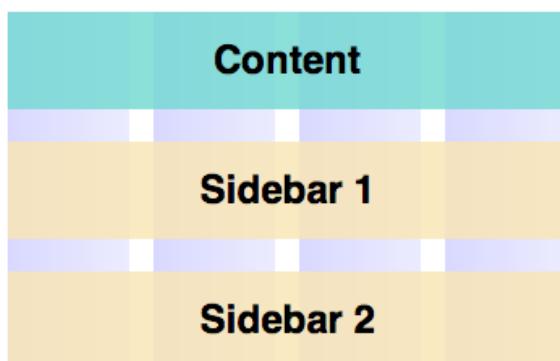
It doesn't matter what your reason is for not using a fluid layout, Susy has you covered with static layouts.

Let's do a deep dive into this topic. You'll learn:

- What is a static layout
- How to create a responsive container for a static layout
- How to lay elements out in a static layout

We're creating a layout for this portion of the book, as usual. This is how it looks like on mobile, tablet and desktop.

Small



Medium



Large



Taking Care of CSS

Lets take care of the CSS used for this layout first.

```
// Taking care of CSS
.content {
  background: rgba(113, 218, 210, 0.8);
}

.sidebar1, .sidebar2 {
  background: rgba(250, 231, 179, 0.8);
}

h2 {
  padding: 1rem 0;
  text-align: center;
}
```

Writing the HTML

The HTML is also very simple, with only main divs.

```
<div class="wrap">
  <div class="content"><h2>Content</h2></div>
  <div class="sidebar1"><h2>Sidebar 1</h2></div>
  <div class="sidebar2"><h2>Sidebar 2</h2></div>
</div>
```

Static Grids in Susy

Static layouts, as its name suggests, are layouts that do not flow according to the browser width. The characteristics of such a layout is that the `width` property of the container is a unit other than percentage.

This `width` changes at each breakpoint to fit the flow of the website. For example:

```
.container {  
  width: 400px;  
  @include breakpoint(700px) {  
    width: 600px;  
  }  
  
  @include breakpoint(1100px) {  
    width: 900px  
  }  
}
```

Susy brings it up a notch when creating static grids. It calculates column and gutter widths in the unit declared.

Here's an example of an output created by a Susy `span()` in a static grid.

```
.testing {  
  width: 135px;  
  float: left;  
  margin-right: 15px;  
}
```

Now that we know how a static grid looks like in CSS, let's move on and create the grid with Susy by first writing the `$susy` map.

The Susy Map

We need to set `math` to static to tell Susy to create static grid units for all of its calculations. When setting `math` to static, we also need to declare a `column-width` key and ensure that the `container` is set to auto.

```
$susy: (
  math: static,
  container: auto,
  column-width: 90px,
);
```

The `column-width` key determines the size of a column. The unit used in this key will also be used for calculating all other widths in the layout.

We have to set `container` to auto when working with `static` grids because Susy will automatically calculate the width of the container for us.

In addition to these 3 keys, We have to give the `$susy` map additional settings to tweak the layout to what we need.

We know that the layout has 4 columns in the mobile view and that the gutter has a size of 15px. Here's the full map we will use for this layout:

```
// Scss
$susy: (
  columns: 4,
  column-width: 90px,
  gutters: 15px / 90px, // or 0.166666667
  math: static,
  global-box-sizing: border-box,
  debug: (image: show)
);

@include border-box-sizing;
```

Let's declare the variables for the breakpoints we will use in this layout as well:

```
// Scss
$bp-med: 845px;
$bp-large: 1245px;
$med: 8;
$large: 12;
```

As always, we have to declare the container. Let's take a look at the output for this `container` mixin.

```
// Scss
.wrap {
  @include container;
}
```

```
/* CSS */
.wrap {
  width: 405px;
  margin-left: auto;
  margin-right: auto;
}
```

Note that `.wrap` has a width of 405px attached to it instead of the max-width of 100% we normally get. Susy has calculated the `width` of this container for us according to the columns, gutters and column-width setting declared in the `$susy` map.

We can verify the accuracy of this output by calculating the width ourselves. There are 4 columns and 3 gutters in the mobile and the math is $(4 \times 90) + (3 \times 15)$, which adds up to 405px.

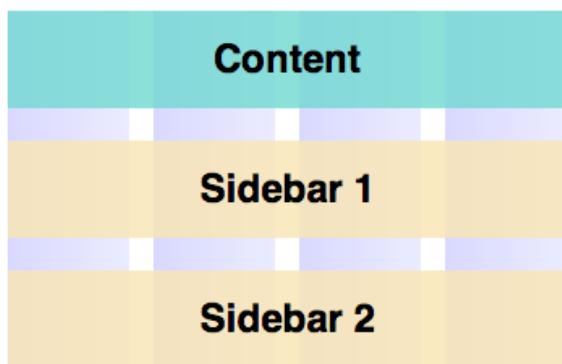
Once we set the container up, we can start laying out the elements of the grid.

[View Source code](#)

Laying It Out With Susy

Let's begin by creating the small layout.

Small Layout



Well, since every element spans the full width, we don't have to do anything to create the small layout :)

Medium Layout



The medium layout takes more work to set up compared to the small one. First of all, notice that we need to change the number of columns from 4 to 8 in the medium layout.

Let's first examine what we need to do for the container.

```
.wrap {  
  @include container;  
  @include susy-breakpoint($bp-med, $med) {  
    width: span($med);  
    @include show-grid;  
  }  
}
```

Since we know the width of `.wrap` is constrained at 405px from the small layout, we need to change the width of the container to match the width of an 8 column layout. This is why we need to create the width with the `span()` function.

Laying the rest of the elements out should be straightforward after coming so far into the book.

```
.content {  
  @include susy-breakpoint($bp-med, $med) {  
    @include span(4);  
  }  
}  
  
.sidebar1 {  
  @include susy-breakpoint($bp-med, $med) {  
    @include span(2);  
  }  
}  
  
.sidebar2 {  
  @include susy-breakpoint($bp-med, $med) {  
    @include span(2 last);  
  }  
}
```

And we get the layout we want.



[View Source code](#)

Large Layout



The large layout is practically a rinse and repeat of creating the medium layout. Since the large layout has 12 columns, we have to rework the container.

```
.wrap {  
  @include container();  
  @include susy-breakpoint($bp-med, $med) {  
    width: span($med);  
    @include show-grid;  
  }  
  @include susy-breakpoint($bp-large, $large) {  
    width: span($large);  
    @include show-grid;  
  }  
}
```

The rest of the layout should be straightforward.

```
// Scss
.content {
  @include susy-breakpoint($bp-med, $med) {
    @include span(4);
  }
  @include susy-breakpoint($bp-large, $large) {
    @include span(8)
  }
}

.sidebar1 {
  @include susy-breakpoint($bp-med, $med) {
    @include span(2);
  }
  @include susy-breakpoint($bp-large, $large) {
    @include span(2)
  }
}

.sidebar2 {
  @include susy-breakpoint($bp-med, $med) {
    @include span(2 last);
  }
  @include susy-breakpoint($bp-large, $large) {
    @include span(2 last)
  }
}
```

And with this code, we get the full layout we wanted.



[View Source code](#)

A Quick Wrap Up

We have covered how to create a static grid with Susy in this chapter. The initial setup for a static grid requires you to set 3 settings in the `$susy` map. They are :

- `math to static`,
- `container to auto`
- `column-width to a length of your choice.`

In addition to the `$susy` map, you'll have to take note to change the width of the container and the number of columns of the grid at each breakpoint.

We have covered a whole lot in this book and you got introduced to a few more settings for the `$susy` map. In the next chapter, let's have a rundown of all the settings that Susy makes available for us.

Susy Settings

We learned a lot about Susy in the last 16 chapters and we even covered how to build an asymmetric grid. You've always used a `$susy` map in every chapter and we only configured it slightly for each of the chapters to get the layouts we wanted. So far, we only managed to cover 9 of the 13 available settings.

In this chapter, we're going to explore all 13 settings fully so you can have an idea on how powerful Susy can be.

You'll learn:

- What the 13 Susy settings are
- How to change Susy settings
- How to use each Susy setting

You will be familiar with all of these settings to a point where you'll never have to try and figure them out from looking through the documentation again. :)

Let's have a look at all 13 settings at their default states before we move on.

List of All 13 Susy Settings

```
// Scss
$susy: (
  flow: ltr,
  math: fluid,
  output: float,
  gutter-position: after,
  container: auto,
  container-position: center,
  columns: 4,
  gutters: .25,
  column-width: false,
  global-box-sizing: content-box,
  last-flow: to,
  debug: (
    image: hide,
    color: rgba(#66f, .25),
    output: background,
    toggle: top right,
  ),
  use-custom: (
    background-image: true,
    background-options: false,
    box-sizing: true,
    clearfix: false,
    rem: true,
  )
);
```

Changing Susy Settings

Changing Susy settings is an easy task. We have done so already a couple of times in the past. You can create a new `$susy` map and insert the settings you want to change. Susy will then take your settings and overwrite the the valaues in the default `$susy` map.

Say you created this `$susy` map for the project.

```
// Scss
$susy (
  output: isolate,
  columns: 1 2 3 4 5,
  gutters: 0.5,
  global-box-sizing: border-box
);
```

Susy will use the map you created and overwrite whatever that was in the default `$susy` map. The result of that is:

```
// Scss
$susy: (
  flow: ltr,
  math: fluid,
  output: isolate, // This has changed
  gutter-position: after,
  container: auto,
  container-position: center,
  columns: 1 2 3 4 5, // This has changed
  gutters: .5, // This has changed
  column-width: false,
  global-box-sizing: border-box, // This has changed
  last-flow: to,
  debug: (
    image: hide,
    color: rgba(#66f, .25),
    output: background,
    toggle: top right,
  ),
  use-custom: (
    background-image: true,
    background-options: false,
    box-sizing: true,
    clearfix: false,
    rem: true,
  )
);
```

Another way to apply the same changes is to create a map of the setting you'd like to use, and insert it by using the `layout()` mixin, just like how we made the `$small` layout while making the asymmetric grid.

Now that we know what happens behind the scenes whenever you change the Susy settings, let's explore all 13 settings!

Exploring All 13 Susy Settings

It's much easier to explain the effects each of the settings has on the eventual output if we had a simple layout work with. Because of this, we will create a small project as we work through this chapter.

The HTML for this small project is the same as the very first layout we created at the beginning of the book:

```
<!-- HTML -->
<div class="wrap">
  <main class="content"><h2>Content</h2></main>
  <aside class="sidebar"><h2>Sidebar</h2></aside>
</div>
```

The additional CSS for this project is:

```
// Scss. (Taking care of CSS)
.content {
  margin-top: 10vh;
  background: rgba(113, 218, 210, 0.8);
}

.sidebar {
  margin-top: 10vh;
  height: 40vh;
  background: rgba(250, 231, 179, 0.8);
}

h2 {
  padding: 1rem 0;
  text-align: center;
  color: #555;
}
```

We will have to create the `$susy` map as usual when we begin this project. Let's only tweak the `debug` key to show the background grid so we know what's going on when we change the layout.

```
// Scss
$susy : (
  debug: (image:show)
);
```

We're now ready to tackle these settings one by one.

From this point on, I'll introduce you to each of the settings and show you the possible configurations in this manner:

```
// Scss
$susy :(
  key: default (option 1 | option 2 | option 3...)
);
```

`key` refers to the key for the map, like `columns` or `global-box-sizing`. `default` refers to the default setting for Susy for the key mentioned, while other possible options are listed with brackets `()` and each option is separated by a `|` separator.

Let's begin with the first setting on the list!

Flow

Flow determines the direction which elements are being stacked. It only has two options, either `ltr` (left to right) or `rtl` (right to left).

```
// Scss
$susy: (
  flow: ltr (ltr | rtl)
);
```

A `ltr` flow means that elements will be floated to the left, and thus stacked from the left to the right.

The reverse is true for `rtl`. Elements will be floated right and are stacked from the right to the left.

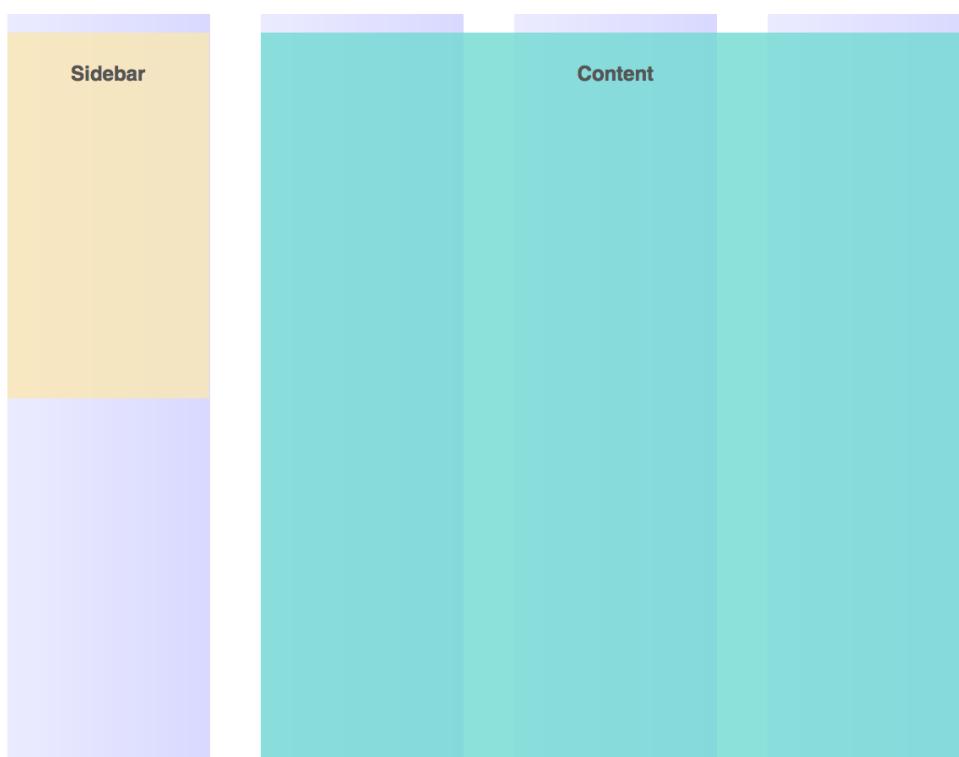
Flow is used if you need to work with languages with text that reads from right to left, and you have to change the design to match accordingly.

If we set the flow to `rtl` and give it the following scss:

```
// SCSS
.content {
  @include span(3 of 4);
}

.sidebar {
  @include span(1 of 4 last);
}
```

The output for this is:



And as you can see, last is now on the left instead of the right.

Summary: `flow` changes the stacking direction of your elements to either left to right or right to left. The only time you'll need to mess around with flow is when you're building a website that requires you tweak the flow of the website to fit the flow of the text.

Math

Math determines whether Susy outputs your CSS values in percentages or in the unit you specified for the `column-width` setting. It has two possible options:

```
// Scss
$susy: (
  math: fluid (static | fluid)
);
```

When set to `math: fluid`, Susy will output all calculated values in percentages.

When set to `math: static`, Susy will calculate the values for containers and gutters according to the value of the `column-width` key. We have covered the Math setting extensively in chapter 17.

Summary: It's safe to set math to `fluid` unless you prefer setting exact units for the layouts you create.

If you go for `static`, make sure to set the `column-width` and to leave the `container` key to auto.

Output

Output determines the layout technique used by Susy to create different layouts. There are only two options right now:

```
// Scss
$susy: (
  output: float (float | isolate)
);
```

`float` is by far the most common layout technique on the web and we have been using it consistently in the book. All elements are floated to the left and

the final element is floated to the right to combat sub-pixel rounding errors. (unless you change the respective Susy setting).

`isolate` is another technique to lay elements out and deals with sub-pixel rounding errors effectively. This technique has been discussed extensively in chapter 14 of the book.

Summary: `float` is the most common and simplest way to layout on the web. Use `isolate` instead if you happen to need precise placements to lay your elements out or if you need to manage sub-pixel rounding issues.

Gutter-position

Gutter position determines where the gutter should be placed in relation to the columns. It also specifies whether these gutters are created as margins or as paddings. Gutter position has 5 different options.

```
// Scss
$susy: (
  gutter-position: after (before | after | split | inside |
  inside-static )
);
```

We have covered gutter positions in detail in chapter 13. This section will be a quick summary of all the available positions.

In `after`, the gutter is created after the column as a `margin-right`. You need to remove the gutter for the final item in the row.

In `before`, the gutter is created before the column as `margin-left`. You need to remove the gutter for the first item in the row.

In `split`, the gutter is split into two and placed on both sides of the column as `margin`. You need to use the `nest` keyword when spanning parent elements.

In `inside` or `inside-static`, the gutter is split into two and placed on both sides of the columns as `padding`. You need to use the `nest` keyword when spanning parent elements.

Summary: Gutter positions affect how you would choose to write your layouts with Susy. Different gutter positions have different quirks and they need to be understood if you want to use them effectively.

There are two major patterns here:

- `before` and `after` outputs gutters to one edge and these gutters have to be removed at the extreme edges.
- `split`, `inside` and `inside-static` splits the gutters up into two and outputs them on either side of the columns. Gutters at the sides do not need to be removed for these positions.

Container

Container sets the `max-width` of the container. It takes in either a length or the value `auto`.

```
// Scss
$susy: (
  container: auto (<length> | auto)
);
```

When set to a specific `length`, like 1140px, Susy will constrain the Susy container with a `max-width` property with the stated length. You are allowed to use any valid CSS unit as the length.

When set to `auto`, Susy will allow the container to have a `max-width` of `100%` if you set the `math` to `fluid`. It will calculate the `max-width` of the container if you set `math` to `static`.

Summary: `container` sets the maximum width of the container. Set this to your preferred `max-width` if you are working with `fluid` math and leave it as `auto` if you are using `static` math.

Container-position

Container position determines whether the container is flushed to the left, centered or flushed to the right of the page.

To be more precise, container position sets the `margin-left` and `margin-right` property for the Susy container.

It takes in 4 different arguments.

```
// Scss
$susy: (
  container-position: center (left | center | right | length )
);
```

The default `center` position will always cause the Susy container to be centered.

```
.container {
  margin-left: auto;
  margin-right: auto;
}
```

The `left` position will cause the container to be flushed to the left of the browser by setting margin-left to 0.

```
.container {  
  margin-left: 0;  
  margin-right: auto;  
}
```

The `right` position will cause the container to be flushed to the right of the browser by setting margin-right to 0.

```
.container {  
  margin-left: auto;  
  margin-right: 0;  
}
```

The `length` argument means that you can give the container-position key a length with any valid CSS unit. This length will be given to both `margin-left` and `margin-right`.

```
// Input  
$susy: (  
  container-position: 50px  
);  
  
.container {  
  @include container;  
}  
  
// Output  
.container {  
  /* ... */  
  margin-left: 50px;  
  margin-right: 50px;  
}
```

Summary: `container-position` helps you align the Susy container to the left, center, right or any specific value you desire. Use whenever you need to.

Columns

Columns defines whether you are using a symmetric or asymmetric grid. At the same time, it will also define the number of columns in your layout.

```
// Scss
$susy: (
  columns: 4 (number | list)
);
```

A `number` causes Susy to output a symmetric grid.

A `list` causes Susy to output an asymmetric grid.

Summary: `columns` can either output a symmetric grid or an asymmetric grid, depending on whether you use a number or a list as its value.

Gutters

Gutters determine the size of the gutters of the grid. You can leave the gutters as a ratio or a fraction.

```
// Scss
$susy: (
  gutters: 0.25 (ratio)
);
```

The ratio given in the gutters setting is the ratio of `gutter-width` divided by `column-width`. If the width of the gutter is 20px and the width of the column is 80px, then a ratio of `20/80` or `0.25` is given to the gutters setting.

The best part about the `gutters` setting is that you can leave the value in fractions when setting it. You can even leave the units intact within this setting, and it will automatically convert correctly.

```
// Scss
$susy : (
  // This automatically translates into 1/4, or 0.25.
  // Note: You need to convert into a decimal ratio if you are
  working with LibSass
  gutters: 20px / 80px;
)
```

When setting gutters in asymmetric grids, make sure to calculate the gutter ratio according to the column width that has a multiple of 1.

One more thing. You can set this value to 0 to remove gutters from the layout.

Summary: Gutters are calculated with a ratio of `gutter / column-width`. If you want to remove gutters, set it to 0.

Column Width

Column width is used to set the exact width of a single grid column. It has two options and is often used in conjunction with a `static` grid.

```
// Scss
$susy: (
  column-width: false (<length> | false)
);
```

A `column-width` of `false` tells Susy to calculate the size of the columns depending on the gutter ratio and number of columns in the layout. This column width will be in percentages.

If a `column-width` is given, Susy will use this `column-width` to calculate the size of the container. It is especially important to set the `column-width` when using a `static` grid.

Summary: Set this if you're using `math: static`. Otherwise, you can safely ignore this setting.

Global Box Sizing

Global box sizing tells Susy what `box-sizing` property should be used for the layouts. There are two available options to choose from.

```
// Scss
$susy: (
  box-sizing: content-box (content-box | border-box)
);
```

We have covered the box sizing and the box model extensively in chapter 4. Flip back if you need a refresher on this setting.

Summary: You have the freedom to choose the box sizing property you use for your website. I recommend sticking with border-box.

Last Flow

Last Flow determines the float direction for the last element in the row. This only affects the `after` gutter position.

```
// Scss
$susy: (
  last-flow: to (from | to)
);
```

`to` is the default that mitigates sub-pixel rounding errors by floating the final element in the opposite direction from the rest of the elements in the row.

In an `ltr` setting, `last` would float the element to the right instead of the left. In an `rtl` setting, `last` would float the element to the left instead of the right.

`from` tells Susy to float the last item in the same direction as the rest of the floated items instead. In an `ltr` setting, the last item will be floated left and in an `rtl` setting, the last item will be floated right.

Unless you really want to expose yourself to sub-pixel rounding errors, there really isn't any reason for you to touch this setting :)

Summary: There's no real reason to change this setting at all!

Debug

Debug determines how the background grid is created for debugging Susy layouts. We have used it heavily over the last few chapters.

Debug is a map that consists of various options. The options are:

```
$susy: (
  debug: (
    image: hide, (hide | show | show-columns | show-baseline)
    color: rgba(#66f, .25),
    output: background, (background | overlay)
    toggle: top right, (left | right and top | bottom)
  )
);
```

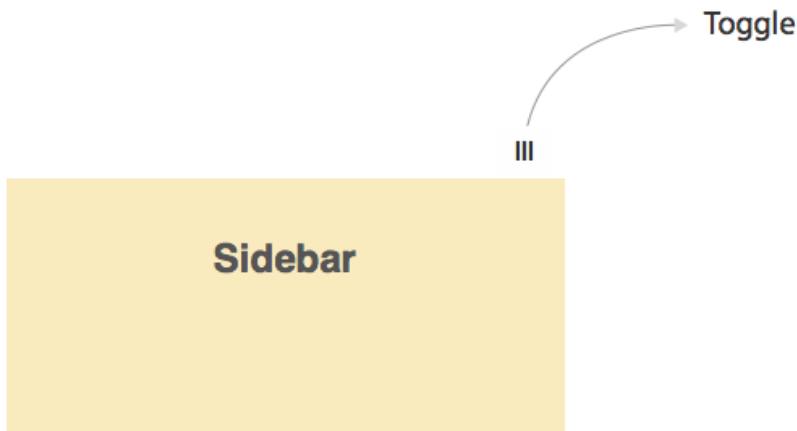
The `image` key determines how Susy will show the background grid.

- `hide` tells Susy to hide the background grid.
- `show` tells Susy to show both columns and baseline.
- `show-columns` tells Susy to only show the columns.
- `show-baseline` tells Susy to only show the baseline.

Susy will only show the baseline if you have Compass Vertical Rhythm installed.

`color` determines the color of the columns in the background grid.

`output` allows you to change the background grid into an `overlay` that can be toggled on or off. When set to `overlay`, a `toggle` will be created and the background grid will only be shown when you hover over this toggle.



`toggle` allows you to change the location of that toggle.

Summary: `debug` is a setting that is mainly used for development and lets you customise how you would like the grid to appear. If you're happy with the background colours, then it's just a matter of setting it to `show` or `hide` and whether to output as `overlay` or `background`.

Use Custom

This setting allows you to use your own mixins for some of Susy's operations.

```
// Scss
$susy: (
  use-custom: (
    background-image: true,
    background-options: false,
    box-sizing: true,
    clearfix: false,
    rem: true,
  )
);
```

Each of these settings is a boolean value. If set to `true`, Susy will attempt to locate a mixin of that same name that is declared in your stylesheet. Failing which, it will default to whatever that is already built into Susy.

`background-image` and `background-options` affect Susy's background grid. You can customise your own `background-image` if you want to.

`box-sizing`, `clearfix` and `rem` affects the output in certain ways. Even if you choose to use a custom mixin, the resultant output is likely to have negligible changes.

Summary: `use-custom` allows you to change some mixins used by Susy, but there are negligible effects on the actual output. To be completely honest, you will probably not even need to touch this setting at all.

A Quick Wrap Up

Now you have an idea of all the 13 settings that Susy provides and how they may potentially affect how you write your code. Most of these settings can be safely ignored, which should be evident from the fact that we haven't been using them so far.

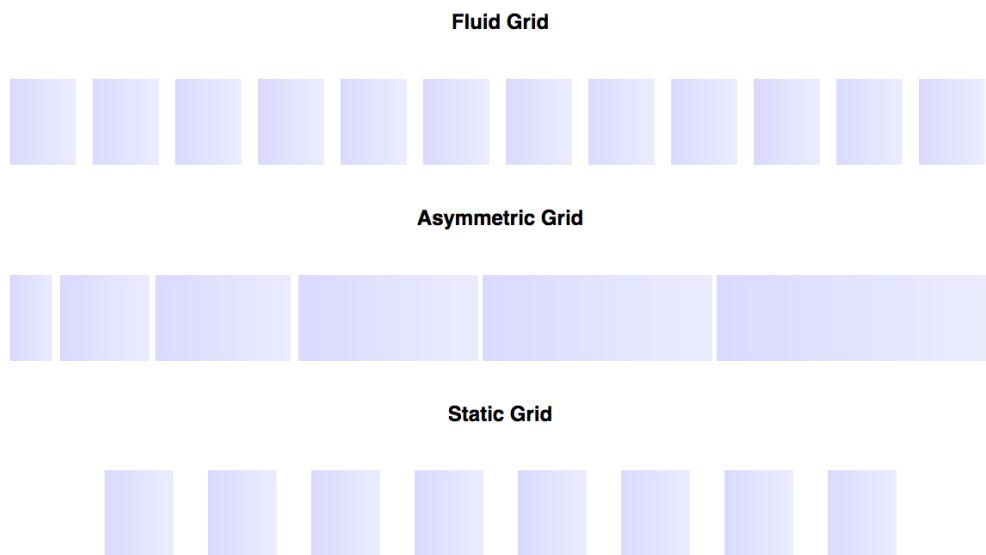
However, their presence allows you to customize Susy to become even more powerful. Here's a quick checklist to help you out when you write the `$susy` map.

1. Determine if you need to change settings that affect how you layout with Susy. These settings are:
 - `output`
 - `gutter-position`
2. Determine the settings for the Susy layout. These settings are:
 - `container`
 - `container-position`
 - `columns`
 - `gutters`
 - `column-width`
 - `global-box-sizing`
3. Determine if you need to tweak any other settings for debugging or unique cases. These settings are:
 - `flow`
 - `respond-to`
 - `last-flow`
 - `debug`
 - `use-custom`

Multiple Susy Grids

We only covered using one Susy grid layout per page so far in the book. When it comes to more complex websites, it might become necessary to include more than one Susy grid.

That's what we're doing in this chapter. We will create a page where we use the 3 types of Susy grids we learnt in the book (Fluid, Asymmetric and Static) on the same page.



You'll learn:

- What are Global and Local scopes in Susy
- How to use these scopes to create multiple Susy grids
- How to change any grid on the fly with local scopes

We first have to understand how global and local scopes work to make multiple grids on the same page. Let's jump right in.

Global and Local Scopes

Sass variables can be defined globally or locally. When a variable is declared outside of a selector, we call it a global variable, i.e. in the global scope.

```
// Scss
// This is a global variable
$var: red;
```

Global variables can be used anywhere within the Sass file as long as the variable has been declared. In this case, `$var` will take on the value of red after `$var: red;` is declared.

```
$var: red;
@debug $var; // returns red

.highlight {
  color: $var; // outputs color: red;
}
```

When the variable is defined within a selector, we say that it is a local variable, i.e. declared within a local scope.

```
// Scss
// This is a local variable
.testing {
  $var: green;
}
```

A local variable will only take on values within the selector that it is defined in. In this case, `$var` will only take on the value of `green` within `.testing`. `$var` will not be defined if used outside of `.testing`, which will result in an error.

```
// Scss
.testing {
  $var: green;
  @debug $var; // returns green
}

@debug $var; // Undefined variable: "$var"
```

We can create a global scope in Sass to take on a default value. When we want to use a more specific value within a certain selector, we can overwrite this global variable with a local one.

```
// Scss

// Declares '$var' to be 'red' globally.
$var: red;
@debug $var; // returns red

.overwritten {
  // Overwrites global '$var' to be 'green' locally.
  $var: green;
  @debug $var; //returns green
}

// $var returns to being red because it is outside of the scope
// where '$var' is overwritten by 'green' in .overwritten.
.testing {
  @debug $var; // returns red
}
```

This is how variable scoping works in Sass. Once we understand the principles behind global and local scoping, we can now understand how global and local scopes work in Susy.

Global and Local Scopes with Susy

The global variable in Susy is the `$susy` map. Once the `$susy` map is defined, it is used anywhere when a Susy function or mixin is used.

```
// Scss
// Susy map is defined to have 12 columns
$susy: (
  columns: 12
);

// Context of span is 12 columns
.span {
  @include span(3);
}
```

We have defined local variables within Susy by using mixins. There are two ways to define local scopes with Susy.

The first way to define local scopes with Susy is to declare Susy settings within the `span()` mixin.

```
// Scss
$susy: (
  columns: 12
);

// Context of span is 9 instead of 12.
.span {
  @include span(3 of 9);
}
```

In the example above, we have effectively created a local `$susy` variable within the `span()` mixin since the global context of `columns: 12` was replaced with 9 columns only within the `span()` mixin.

Coding this way causes us to repeatedly declare similar local scopes, which is not ideal.

The second method of defining local scopes in Susy is to use the `nested()` mixin. This method is much cleaner compared to the previous one as it extends the local scope and allows us to reuse it multiple times, if needed.

```
// Scss
$susy: (
  columns: 12
);

// Local scope of 9 columns apply to everything within the
// nested mixin
@include nested(9) {
  @include span(3); // same as @include span(3 of 9)
  margin-bottom: gutter(); // same as gutter(9)
}
```

As you can see above, the `columns` setting within the Susy map has changed from 12 to 9 for the Susy mixin used within this `nested()` mixin.

We can apply the same method to create multiple grids. However, we use the `with-layout()` mixin instead of the `nested()` mixin.

Using Multiple Grid Layouts

The `with-layout()` mixin takes in a map argument and overwrites the `$susy` map with every setting given to it. When this happens, we create a local `$susy` map for every `with-layout()` mixin used.

Within this `with-layout()` mixin, we can create different grids. Let's put that in an example.

Say we want to create a page with 3 different grids. A fluid grid, an asymmetric grid and a static grid. The starter HTML and CSS for this page is:

```
<!-- Html -->
<section>
  <h2>Fluid Grid</h2>
  <div class="fluid-wrap"></div>
</section>

<section>
  <h2>Asymmetric Grid</h2>
  <div class="asym-wrap"></div>
</section>

<section>
  <h2>Static Grid</h2>
  <div class="static-wrap"></div>
</section>
```

```
// Scss
h2 {
  text-align: center;
  padding: 2rem;
}

.fluid-wrap,
.asym-wrap,
.static-wrap {
  height: 100px;
}
```

Since we have multiple grids on the same page, it would make sense to create the `$susy` map with settings that apply to all 3 grids. In this case, we will set the `global-box-sizing` and `debug` keys.

```
// Scss
$susy: (
  global-box-sizing: border-box,
  debug: (image: show)
);

@include border-box-sizing;
```

Next, create 3 different maps by varying their columns, gutters and other necessary settings.

```
$fluid-grid: (
  container: 1140px,
  columns: 12,
  gutters: 0.25
);

$asym-grid:(
  output: isolate,
  container: 1140px,
  columns: 1 2 3 4 5 6,
  gutters: 0.1

);

$static-grid: (
  math: static,
  column-width: 80px,
  columns: 8,
  gutters: 0.5
);
```

To create the different grids, we can wrap the `.wrap` selectors with the `with-layout()` mixins.

```

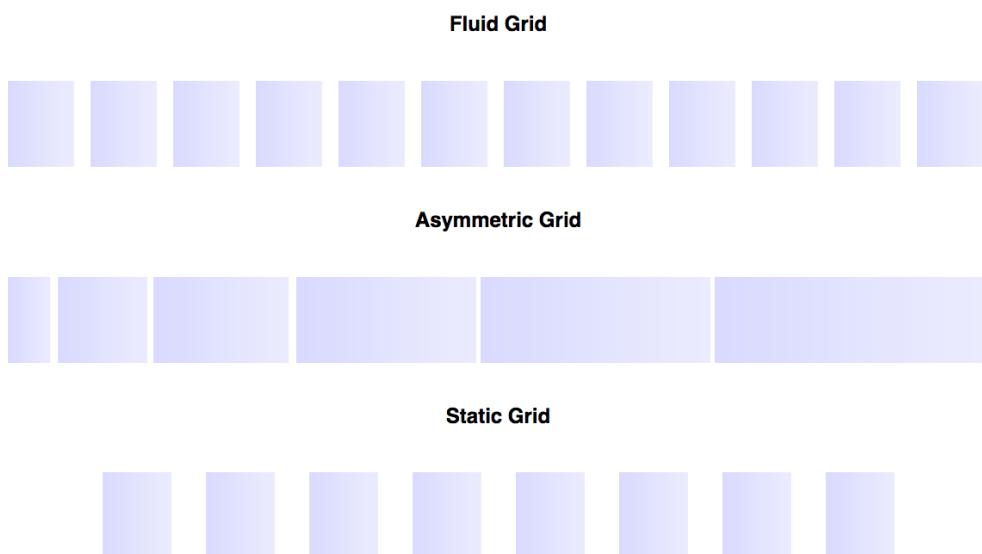
.fluid-wrap {
  @include with-layout($fluid-grid) {
    @include container;
  }
}

.asym-wrap {
  @include with-layout($asym-grid) {
    @include container;
  }
}

.static-wrap {
  @include with-layout($static-grid) {
    @include container;
  }
}

```

And now, we have 3 different grids laid out on the same page!



[View Source code \(tag 19-1\)](#)

A Quick Wrap Up

We covered how to use multiple grids within the same page in this chapter. The whole idea is to use a `with-layout()` mixin to create a local scope for each Susy grid that you'd like to use.

One more thing.

What if you needed a way to test a grid before creating a map for it? With Susy, you can! Here's a sneak peek of what we will be covering in the next chapter.

Say we want to create a different layout that has 8 columns, 0.5 gutters and a split gutter position, we can do so with the following code:

```
// Scss
.onthefly1-wrap {
  @include with-layout(8 0.5 split) {
    @include container;
  }
}
```

On The Fly!



This is called the `shorthand` syntax. We will explore this in the next chapter.

Susy Shorthand

The Susy shorthand allows you to quickly overwrite Susy global settings with local settings of your choice. The option to use the Susy shorthand works with any Susy mixin or function.

We have encountered the shorthand multiple times when we were just beginning to learn to code with Susy. The most basic form of the shorthand is:

```
@include span(1 of 4);
```

I also showed you a more complex shorthand syntax in the last chapter:

```
@include with-layout(12 0.5 split) {  
    // stuff...  
}
```

We will demystify the shorthand syntax in this chapter.

You'll learn

- What options the shorthand syntax provides
- How to write the shorthand
- Things to watch out for when writing the shorthand

Shall we begin?

Shorthand Syntax

The shorthand syntax is a shortcut for you to overwrite settings that are declared in the `$susy` map. It also provides extra arguments to help us create our layouts.

The shorthand syntax can be summarised into a series of arguments:

```
// Scss  
shorthand: $span of $grid $location $keywords
```

Each set of arguments has its own rules. We will cover each set step by step, beginning with `$span`.

Span

`$span` refers to the width of the element that we're creating. It can take in a length unit or a unitless number.

If a unitless number is used, Susy will use this number as the number of columns like we have been using so far.

```
.span {  
  @include span(3); // refers to 3 columns  
}
```

If a length unit was used, Susy will use this inserted value as the width of the element.

```
.span {  
  width: span(25%); // this will output 25%  
  width: span(100px); // this will output 100px  
}
```

You will find yourself using the unitless number to span columns most of the time. I never found a good reason to use the unit values when working with span.

Quick Summary: Span refers to the width of the element you are trying to create. Most of the time, you'll just use unit-less numbers to signify the number of columns to span.

Grid

`$grid` refers to the columns and gutters of the grid you are creating. It is a combination of `columns`, `gutters` and `column-width` settings.

`$columns` is a required argument when writing the `$grid` keyword, while `$gutters` and `$column-width` are optional arguments. `$gutters` and `column-width` will use the settings within the global `$susy` map if they are not passed to the mixin.

In its most basic form, the `$grid` takes on the following syntax:

```
$grid: $columns $gutters;

// 12-column grid with default gutters
$grid: 12;

// 12-column grid with 1/3 gutter ratio
$grid: 12 1/3
```

When writing the `$grid`, we can use parentheses to tell Susy that certain things are grouped together. This allows us to write asymmetric columns and gives room for us to write the column width.

```
// 12-column grid with 60px columns and 10px gutters
// 60px columns and 10px gutters will be converted into 0.16667
gutters
$grid: 12 (60px 10px);

// asymmetrical grid with 5 columns and a .25 gutter ratio
$grid: (1 2 3 2 1) .25;
```

The `$grid` argument set is always preceded with the `of` keyword when used in a mixin or a function.

```
// Scss
.grid {
  @include span(3 of 4);
}

.grid2 {
  @include span(3 of 6 (20px 10px));
}
```

Quick Summary: `$grid` tells Susy the combination of `columns`, `gutters` and `column-width` settings to use for the local scope.

Location

`$location` refers to the place where the span is supposed to be at. It tells Susy to output the span in a specific location. These locations can be `first`, `last` or `at <number>`.

We used the `first` and `last` keyword while we created the `fluid` grid to control whether the element is the first or last on the row. We have also used the `at <number>` keyword when working with the Isolate Technique.

The usage of `$location` will vary depending on the `gutter-position` and `output` of the Susy settings you have created for the map.

Quick Summary: The `$location` keyword is an important part of Susy, and you have seen it used in many places. There are only 3 values you can give to it - `first`, `last` and `at <number>`.

Keywords

We come to the final set of arguments for the Susy shorthand. There are two different types of keywords we can use.

1. Global keywords
2. Local keywords

Global Keywords

Global keywords are keywords that change the settings within the `$susy` map. This set of keywords are options that you could set within the susy map.

```
container: auto,  
math: static | fluid,  
output: isolate | fluid  
container-position: left | center | right  
flow: ltr | rtl  
gutter-position: before | after | split | inside | inside-static  
debug (image): show | hide | show-columns | show-baseline  
debug (output): background | overlay
```

You can use any number of keyword combinations in your mixins or functions. Keywords that are omitted will get their values from the global `$susy` map.

The only exception is that you can only use the `debug` keywords with the `container()` mixin.

Here are some examples of how the keywords can be used at the same time.

```
.wrap {  
  @include container( auto static right ltr inside );  
}  
  
.span {  
  @include span( isolate 8 of 12 0.5 at 2 split );  
}
```

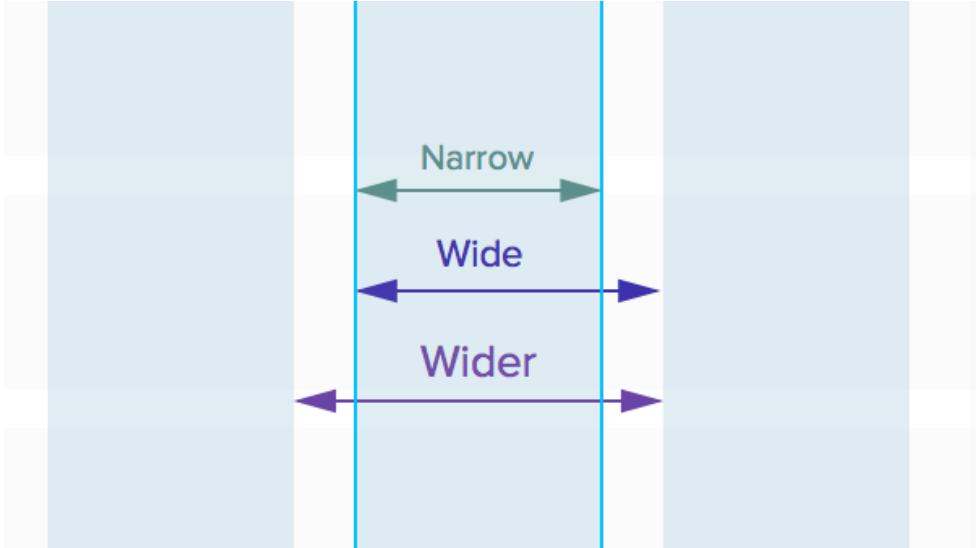
Local Keywords

Local keywords are keywords that provides additional context for the mixin or function to create the correct value. These keywords are optional.

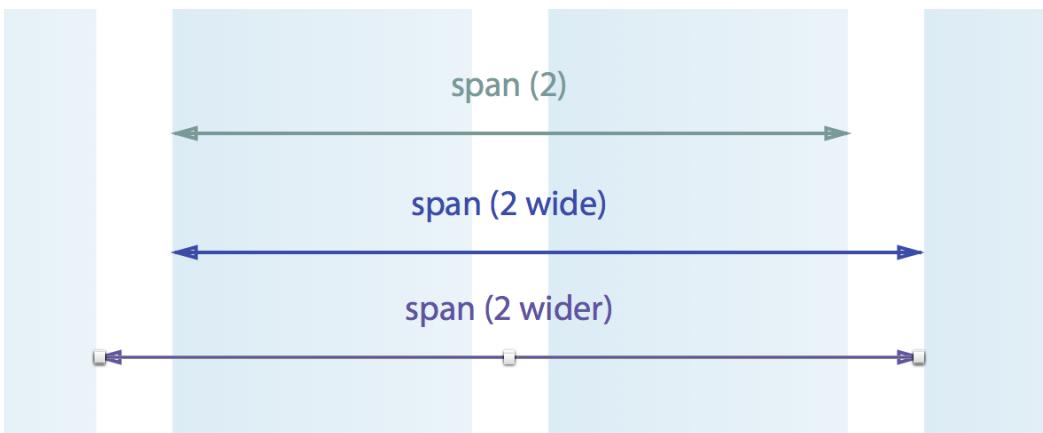
```
spread: narrow (default) | wide | wider,  
role: nest  
clear: break | nobreak,  
gutter-override: no-gutters | no-gutter,
```

`spread` adjusts the width of the columns. There are 3 options where you can choose from:

- Narrow (*default*)
- Wide
- Wider



When set to `narrow`, Susy will include all internal gutters. When set to `wide`, Susy will include the width of one external gutter and it will include the width of two external gutters if set to `wider`.



`nest` tells Susy to set the element as a parent container for a `split` or `inside` grid, and it recalculates the width of the element. It is only applicable for `split`, `inside` and `inside-static` grids.

`clear` has two options – `break` or `nobreak`. `break` adds a `clear: both` property to the output of the `span()` mixin while `nobreak` adds a `clear: none` property to the mixin.

`gutter override` removes the gutters from the grid. It doesn't set gutters to 0, it simply removes all margins and paddings that act as gutters.

A Quick Wrap Up

The Susy shorthand is made up of 4 different argument sets. `$span` , `$grid` , `$location` and `$keywords`. Once you understand the shorthand, you can create any layout easily on the fly without placing them into a map.

However, knowing it doesn't mean you should do it...

```
// Don't do this!
span(isolate 3 at 2 of 4 inside (10em 4em) wider ...)
```

We have covered a ton of Susy knowledge. So far, we know that context is important when it comes to using Susy. If you are working on a larger project, you need to know of a way to handle Susy contexts effortlessly.

That's what we're covering in the next chapter.

Handling Difficult Susy Contexts

The concept of contexts has been covered multiple times within this book and by now, it should be obvious that contexts are extremely important.

Contexts are easy to find if we have a simple website, working with a symmetric grid. They get much harder to locate as the website gets more complicated.

In this chapter, we will explore a method that allows you to identify, store and retrieve any context easily without having to twist your mind and figure it all out.

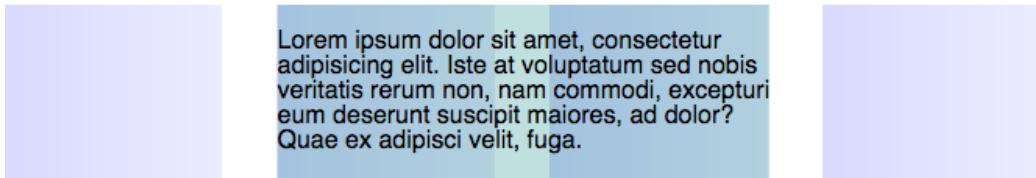
You'll learn

- How to manage contexts
- How to save contexts
- How to use the saved contexts

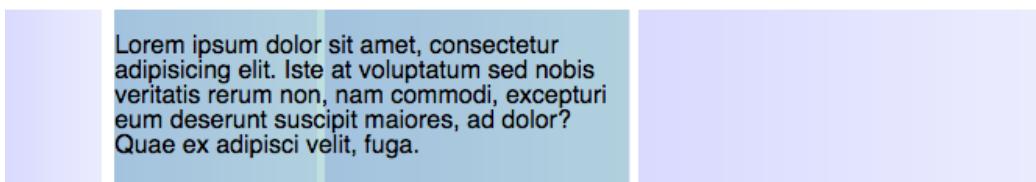
We're creating two layouts in this chapter and we're going to manage their contexts as the breakpoint changes.

Small

Fluid Grid

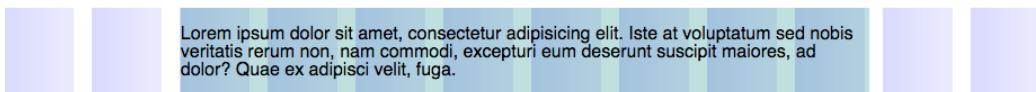


Asymmetric Grid

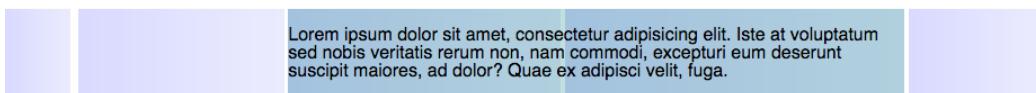


Large

Fluid Grid



Asymmetric Grid



Let's begin, shall we? :)

Taking Care of HTML and CSS

We start off the chapter by adding the necessary HTML and CSS for this layout as usual. The HTML for this layout is similar to the one we made with multiple susy-grids on the same page. This time round, we add a `.span` within each grid and give them some text.

```

<section>
  <h2>Fluid Grid</h2>
  <div class="fluid-grid">
    <div class="span">
      <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Iste at voluptatum sed nobis veritatis rerum non, nam commodi, excepturi eum deserunt suscipit maiores, ad dolor? Quae ex adipisci velit, fuga.</p>
    </div>
  </div>
</section>

<section>
  <h2>Asymmetric Grid</h2>
  <div class="asym-grid">
    <div class="span">
      <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Iste at voluptatum sed nobis veritatis rerum non, nam commodi, excepturi eum deserunt suscipit maiores, ad dolor? Quae ex adipisci velit, fuga.</p>
    </div>
  </div>
</section>

```

This demo CSS is simple. We only need to give the `<h2>` and `.span` some styles.

```

h2 {
  text-align: center;
  margin: 1.5em 0;
}

.span {
  background: transparentize(teal,0.75);
}

```

Now that we have gotten the HTML and CSS out of the way, let's begin looking at the context mixins.

The Context Mixins

We need to create different context mixins because Susy doesn't provide us with a way to manage different contexts easily yet.

Here are 4 mixins we will be using, remember to add them to the top of your Scss file! We will go through how to use the mixins as we use them to create the layout.

```
// Scss
$contexts: () !default;

// With Context [Mixin]
// -----
// Use a nested grid with predefined contexts
// - $keys... : <path-to-contexts>
@mixin with-context(
  $keys...
) {
  $inspect  : _susy-deep-get($contexts, $keys...);
  $context  : parse-span($inspect);
  $old      : susy-get(columns);

  // Checks and sets column to correct nested syntax
  $columns  : if(map-has-key($context, columns),
nested($context), $inspect);

  $susy: map-merge($susy, (columns: $columns)) !global;

  @include susy-inspect(nested, $inspect);
  @content;

  $susy      : map-merge($susy, (columns: $old)) !global;
}

// BP With Context [Mixin]
// -----
```

```

// Use a nested grid with predefined context in a breakpoint
// - $breakpoints : <breakpoint>
// - $keys         : <path-to-context>

@mixin bp-with-context($breakpoint, $keys...) {
  @include breakpoint($breakpoint) {
    @include with-context($keys...) {
      @content;
    }
  }
}

// Span AC [Mixin]
// -----
// uses span mixin and adds context to $context map
// - $span          : <span>
// - $name          : <name-of-identifier>
// - $breakpoint   : <breakpoint>
@mixin span-ac($span, $name, $breakpoint: null) {
  @include span($span);
  @include add-context($span, $name, $breakpoint);
}

// Add Context [Mixin]
// -----
// Adds context to $context map
// - $span          : <span>
// - $name          : <name-of-identifier>
// - $breakpoint   : <breakpoint>
@mixin add-context($span, $name, $breakpoint: null) {
  $map: parse-span($span);
  $n: susy-get(span, $map);
  $columns: susy-get(columns, $map);
  $gutters: susy-get(gutters, $map);
  // $spread: susy-get(spread, $map);
  $edge: get-edge($span);
  $location: get-location($map);
  // Count number of columns if n is 'full'
  @if $n == full or $n == null {

```

```

@for $n -- from 0 to $n -- null {
    $n: susy-count($columns);
}
// Resets edge to null if set to 'full'
@if $edge == full {
    $edge: null;
}
// Adds span to $contexts
@if $breakpoint == null {
    @if $edge {
        $contexts: _susy-deep-set($contexts, $name, $n of $columns
$edge) !global;
    }
    @else {
        $contexts: _susy-deep-set($contexts, $name, $n of $columns
at $location) !global;
    }
}
@else {
    @if $edge {
        $contexts: _susy-deep-set($contexts, $name, $breakpoint,
$n of $columns $edge) !global;
    }
    @else {
        $contexts: _susy-deep-set($contexts, $name, $breakpoint,
$n of $columns at $location) !global;
    }
}
}

```

The Susy Map

Since we're working on two different grids on the same page, let's create the Susy map in a way that all settings used are common to the two grids again.

```
// Scss
$susy: (
  global-box-sizing: box-border,
  debug: (image: show-columns)
);

@include border-box-sizing;
```

We are creating two grids, a fluid grid and an asymmetric grid. Let's add two other maps to hold these two different grids.

```
// Scss
$fluid-grid: (
  container: 1140px,
  columns: 4,
  gutters: 0.25,
);

$asym-grid:(
  output: isolate,
  container: 1140px,
  columns: 1 2 3 4,
  gutters: 0.1
);
```

Next, we can create the grid with the `with-layout()` mixin. These should be familiar to you since we have covered them in Chapter 19.

```
// Scss
.fluid-grid {
  @include with-layout($fluid-grid) {
    @include container;
  }
}

.asym-grid {
  @include with-layout($asym-grid) {
    @include container;
  }
}
```

<%= commit “21-1”, “Created multiple grid backgrounds” %>

There is a breakpoint at `800px` where both grids will show a different grid. Asymmetric grids are more troublesome compared to symmetric grids because we need to pass in a map instead of the usual Susy shorthand (Otherwise things get screwed up).

```

// Scss
$fluid-large: 12;
$asym-large: (columns: 1 2 4 5 2);

$bp-large: 800px;

.fluid-grid {
  @include with-layout($fluid-grid) {
    @include container;

    @include susy-breakpoint($bp-large, $fluid-large) {
      @include show-grid();
    }
  }
}

.asym-grid {
  @include with-layout($asym-grid) {
    @include container;

    // When working with asymmetric grids we cannot use the
    // susy-breakpoint mixin
    @include breakpoint($bp-large) {
      @include nested($asym-large) {
        @include show-grid();
      }
    }
  }
}

```

Fluid Grid

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Iste at voluptatum sed nobis veritatis rerum non, nam commodi, excepturi
eum deserunt suscipit maiores, ad dolor? Quae ex adipisci velit, fuga.

Asymmetric Grid

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Iste at voluptatum sed nobis veritatis rerum non, nam commodi, excepturi
eum deserunt suscipit maiores, ad dolor? Quae ex adipisci velit, fuga.

Now that's a handful to write. The extra complications that come with asymmetric grids aren't fun to deal with either. The additional mixins that we have included at the top of this chapter solves these problems.

Here's how.

Using The Context Mixins

To use the new context mixins, we first need a place to store all our contexts. These contexts are stored in the `$context` map automatically for you.

```
$contexts: () ;
```

When we begin writing the layout, we can add different contexts to the `$context` map, including different contexts for both `fluid-grid` and `asym-grid` at multiple breakpoints.

You can treat asymmetric grids and symmetric grids in the same manner:

```
$contexts: (
  fluid: (
    small: 4,
    large: 12
  ),
  asym: (
    small: 1 2 3 4,
    large: 1 2 4 5 2
  ),
);
```

When you place a context into the `$context` map, you gain the ability to retrieve this context with the `with-context()` mixin that was created above.

```
// Scss
@include with-context($keys...) {
  @content;
}
```

\$keys refer to keys within the map to get to your stored context. In this case, if we want to retrieve the large context from the fluid grid, we can write

```
// Scss
@include with-context(fluid, large) {
  // Properties here...
}
```

With this information, let's rewrite the grid background Sass.

```
// Scss
.fluid-grid {
  @include with-layout($fluid-grid) {
    @include container();
    @include breakpoint($bp-large) {
      @include with-context(fluid, large) {
        @include show-grid();
      }
    }
  }
}

.asym-grid {
  @include with-layout($asym-grid) {
    @include container();
    @include breakpoint($bp-large) {
      @include with-context(asym, large) {
        @include show-grid();
      }
    }
  }
}
```

And you'll get the same result.

Fluid Grid

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Iste at voluptatum sed nobis veritatis rerum non, nam commodi, excepturi
eum deserunt suscipit maiores, ad dolor? Quae ex adipisci velit, fuga.

Asymmetric Grid

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Iste at voluptatum sed nobis veritatis rerum non, nam commodi, excepturi
eum deserunt suscipit maiores, ad dolor? Quae ex adipisci velit, fuga.

It's still kinda clunky to have to write so many include statements. We then created a convenience mixin that combines `breakpoint()` and `with-context()` together - `bp-with-context`

```
@include bp-with-context($breakpoint, $keys...) {  
  @content;  
}
```

`$breakpoints` refer to the media query arguments you give to the `breakpoint()` mixin while `$keys...` refer to the same keys in the `with-context` mixin discussed above. Note that you cannot do `no-query` fallbacks with this convenience mixin.

```
// Scss
.fluid-grid {
  @include with-layout($fluid-grid) {
    @include container();
    @include bp-with-context($bp-large, fluid, large) {
      @include show-grid();
    }
  }
}

.asym-grid {
  @include with-layout($asym-grid) {
    @include container();
    @include bp-with-context($bp-large, asym, large) {
      @include show-grid();
    }
  }
}
```

<%= commit “21-2”, “Grid backgrounds using with-context mixin” %>

Since we managed to get the grid background right, we can now move on to laying the `.span` element out with Susy.

Laying It Out With Susy

When we lay the `.span` element out with Susy, we have to take note of the context once again. We can use the `with-context()` mixin or `bp-with-context()` mixin like how we have done with the grid wrappers.

```
// Scss
.fluid-grid {
  @include with-layout($fluid-grid) {
    // grid-background-styles...

    .span {
      @include span(2);
      margin-left: span(1 wide);
      @include bp-with-context($bp-large, fluid, large) {
        @include span(8);
        margin-left: span(2 wide);
      }
    }
  }
}

.asym-grid {
  @include with-layout($asym-grid) {
    // grid-background-styles...

    .span {
      @include span(2 at 2);
      @include bp-with-context(800px, asym, large) {
        @include span(2 at 3);
      }
    }
  }
}
```

With this, we get both the small and large layouts we set out to get in this chapter.

<%= commit “21-3”, “Create Span for fluid and asym grids” %>

We also want to learn how to add contexts into this `$contexts` map so we can get them for nested grids if we have any.

Adding Contexts

The new `add-context()` mixin we added above allows us to add a context into the `$context` map. It takes in 3 arguments in total.

```
@include add-context($span, $name, [$breakpoint]);
```

`$span` refers to the arguments you place into the `span()` mixin.

`$name` refers to the name of the first key you use in the `$contexts` map. Examples we have used so far are `asym` and `fluid`.

`$breakpoint` refers to the second key within the `$contexts` map. Examples we have used so far are `small` and `large`.

Let's use the fluid grid as our example for adding new contexts into the `$contexts` map.

```
// Scss
.fluid-grid {
  @include with-layout($fluid-grid) {
    // grid-background-styles...

    .span {
      @include span(2);
      // ...
      @include bp-with-context($bp-large, fluid, large) {
        @include span(8);
        // ...
      }
    }
  }
}
```

We have two contexts to add to the `$contexts` map here. A small context and a large context. We will name their `$breakpoints` as `small` and `large`.

Say we name the second key as `fluid-span`. The code to add context is:

```
.fluid-grid {  
  @include with-layout($fluid-grid) {  
    // grid-background-styles...  
  
    .span {  
      @include span(2);  
      @include add-context(2, fluid-span, small);  
      // ...  
      @include bp-with-context($bp-large, fluid, large) {  
        @include span(8);  
        @include add-context(8, fluid-span, large);  
        // ...  
      }  
    }  
  }  
}
```

Once done, we want to check whether this context is indeed added to the `$contexts` map.

```
@debug $contexts;  
  
// Output from debug  
$contexts: (  
  // fluid and asym contexts ...  
  
  fluid-span: (  
    small: 2 of 4 at null,  
    large: 8 of 12 at null  
  )  
)
```

The `fluid-span` contexts for both the small and large breakpoints have been added successfully to the `$contexts` map as you can see from the output.

Note that the syntax is slightly different from the columns syntax we placed in the `fluid` and `asym` contexts. This is because more information needs to be extracted to get the contexts for asymmetric grids (This syntax is how we managed to get a nested context for an asymmetric grid).

Let's do the same for the `asym-grid` as well

```
.asym-grid {  
  @include with-layout($asym-grid) {  
    // grid background styles ...  
  
    .span {  
      @include span(2 at 2);  
      @include add-context(2 at 2, asym-span, small);  
  
      @include bp-with-context($bp-large, asym, large) {  
        @include span(2 at 3);  
        @include add-context(2 at 3, asym-span, large);  
      }  
    }  
  }  
}  
  
@debug $contexts;  
  
// Output from debug  
$contexts: (  
  // ...  
  
  asym-span: (  
    small: 2 of (1 2 3 4) at 2,  
    large: 2 of (1 2 4 5 2) at 3  
  )  
);
```

We now have both `fluid-span` and `asym-span` in the `$contexts` map.

It can be troublesome to write a line of code specifically to add the context. The other flaw here is that the arguments we give to the `span()` mixin is exactly the same as the first argument of the `add-context()` mixin.

We can combine the two into a convenience mixin, the `span-ac()` mixin.

It takes in the same arguments as the `add-context()` mixin and does both jobs. We can now simplify the code to:

```
// Scss
.fluid-grid {
  @include with-layout($fluid-grid) {
    // grid background styles ...

    .span {
      @include span-ac(2, fluid-span, small);
      margin-left: span(1 wide);
      @include bp-with-context($bp-large, fluid, large) {
        @include span-ac(8, fluid-span, large);
        margin-left: span(2 wide);
      }
    }
  }
}

.asym-grid {
  @include with-layout($asym-grid) {
    // grid background styles ...

    .span {
      @include span-ac(2 at 2, asym-span, small);
      @include bp-with-context(800px, asym, large) {
        @include span-ac(2 at 3, asym-span, large);
      }
    }
  }
}
```

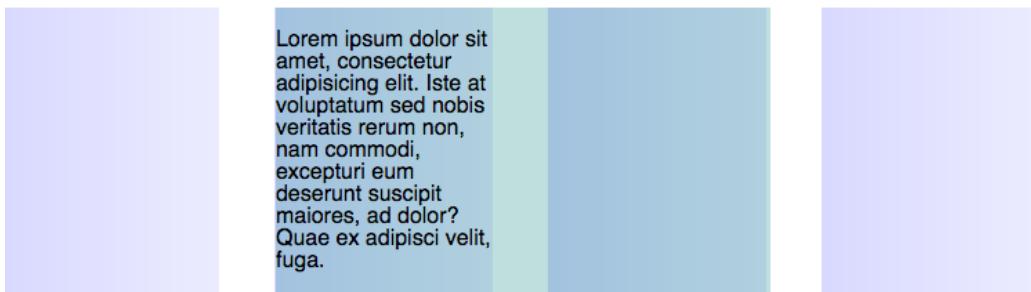
<%= commit “21-4”, “Added contexts to \$contexts map” %>

Once the new contexts are set in the `$contexts` map, we can retrieve and use them anytime using the above method.

Since we’re going to work on the nested contexts, let’s play around with the `<p>` element slightly to test it out.

small

Fluid Grid

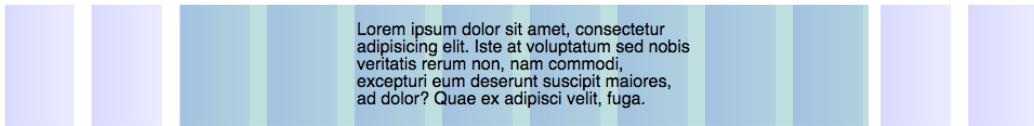


Asymmetric Grid

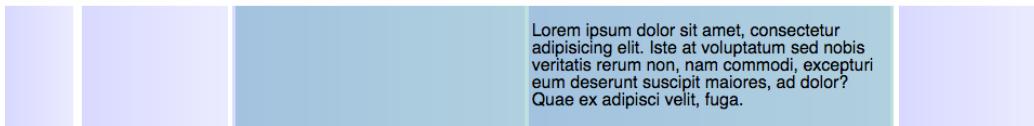


large

Fluid Grid



Asymmetric Grid



Retrieving New Contexts

We retrieve contexts in the same way as how we did with the `.span` mixin earlier. The difference this time is we use the `fluid-span` or `asym-span` instead.

Once we get the context, we can use the `span()` mixin, just like how we use it in a `nested()` mixin.

```
// Scss

.fluid-grid {
  @include with-layout($fluid-grid) {
    // ...

    p {
      @include with-context(fluid-span, small) {
        @include span(1);
      }
      @include bp-with-context($bp-large, fluid-span, large) {
        @include span(4);
        margin-left: span(2 wide);
      }
    }
  }
}

.asym-grid {
  @include with-layout($asym-grid) {
    // ...

    p {
      @include with-context(asym-span, small) {
        @include span(1 first);
      }
      @include bp-with-context($bp-large, asym-span, large) {
        @include span(1 last);
      }
    }
  }
}
```

<%= commit “21-5”, “Retrieve new contexts” %>

A Quick Wrap Up

It's incredibly important to get the context correct when working with Susy. Everything falls in place nicely once the context is correct.

We covered how to work with the `$contexts` map, how to add and retrieve contexts from this map and how to use it for working with nested grid contexts on both symmetric and asymmetric grids.

As you can see, Susy is great at laying out grids. However, it lacks UI elements when compared to traditional frameworks. The good news is you can create your own UI element set and integrate it nicely with Susy. We will explore how in the next chapter.

Integrating Susy With Your UI Kit

UI Kits, or User Interface Kits, are a library of pre-styled components. They also go by the name “Frameworks” in the development world. Popular examples of such frameworks include Bootstrap and Foundation.

Having these UI Kits allow you to plug and play different components into your site’s code, allowing you to prototype a site quickly.

One of the major complaints about Susy is that it does not come with a defined UI kit, and that makes prototyping with Susy slower.

The good news here is that Susy can integrate with any UI kit if you want to. You can even use any UI element from Bootstrap or Foundation within your very own starter kit.

We will be exploring how in this chapter.

You’ll learn

- How UI Kits work
- How Susy integrates with UI kits
- How use Susy with your UI kit of choice

Let’s begin this chapter by gaining a deeper understanding on UI Kits because we can only use Susy with these kits after we know how they work.

How UI Kits work

As mentioned, UI Kits are libraries of components that are pre-styled by the kit developers. This means that they have defined what classes you have to use for both the HTML and CSS.

Predefined Classes

For example, you can style an `<a>` tag into a button with bootstrap by giving it a `.btn` class.

```
<!-- Creating a button with Bootstrap -->
<a class="btn" href="#">I am a button</a>
```

Creating a button with Foundation is similar, except you use `.button` instead of `.btn`.

```
<a class="button" href="#">I am a button</a>
```

You will not be able to get a button in Bootstrap if you give the `<a>` tag a `.button` class like in Foundation.

Additional Classes

Each kit may also come with additional classes for their UI elements to take on different styles.

For example. Bootstrap allows you to style buttons to a “success” state with `.btn-success` while Foundation allows you to do the same by adding a `.success` class.

```
<!-- Success button in Bootstrap -->
<a href="btn btn-success">I am a button</a>

<!-- Success button in Foundation -->
<a class="button success" href="#">I am a button</a>
```

These additional classes are also predefined by the kit builders.

Changing UI Styles

Since it's likely that you would want to customize your site, kit developers usually include a variables file for you to make your personal changes.

This variables file allow you to tweak the colors, margins, paddings and anything else that was used to create the UI Component.

However, because different developers have different preferences, they may choose to name their variables file differently.

For example, Bootstrap stores all their variables in a `_variables.scss` file while Foundation stores their variables in a `_settings.scss` file.

JavaScript Components

JavaScript may be used for more complex components within the UI Kit. These JavaScript components target classes as well, and you must be sure to include the correct classes in your HTML.

Grid Systems

Larger UI Kits like Bootstrap and Foundation come with their own grid systems. These grid systems provide predefined classes like `.col-md-9` in Bootstrap and `small-4` in Foundation.

As you can tell by now, you can use UI Components from any UI Kit in any project as long as you use the correct classes for in your HTML

How Susy Integrates with UI Kits

We know that Susy is a layout engine. It creates the basic layout for your site. Susy will not interact with most UI Components since components are usually not built within layouts.

Say you have a Foundation progress bar and it needs to be 6 columns wide.

```
<div class="progress">
  <span class="meter"></span>
</div>
```

```
.progress {
  @include span(6 of 8);
  margin-left: span(1 wide of 8);
}
```



In short, you can use Susy as you would normally (just ditch the grid) :)

Using Susy with your UI Kit of Choice

Since Susy works with any UI Kit, what you have to do is to add them to the same project. We will use Bootstrap as the main framework we are integrating with for this chapter.

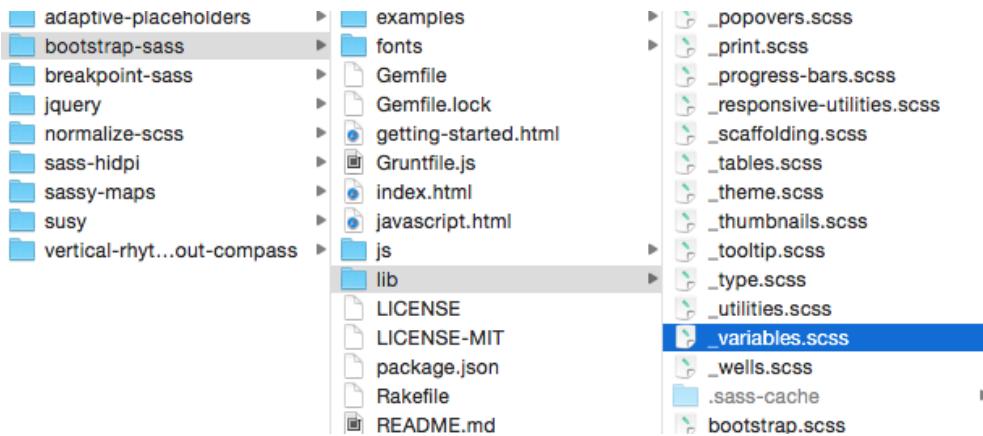
You will first need to add your Bootstrap-sass to your project either by [downloading Bootstrap-Sass](#) or by using Bower.

```
bower install bootstrap-sass --save-dev
```

Create two Sass files in your project and name them `_bootstrap-variables` and `_bootstrap-components`, and import them into your stylesheet.

```
@import "susy";
// Bootstrap stuff
@import "bootstrap-variables"
@import "bootstrap-components"
```

Next, navigate to the `_variables.scss` file within the Bootstrap folder. It should be found under the `lib` folder in the Bootstrap library you downloaded.



Copy the contents of that folder and paste that into your own `_bootstrap-variables.scss` file. From this point onwards, you can feel free to adjust Bootstrap's variables to your preference.

Next, you will have to determine what components you want to add to your project. Navigate to the same `lib` folder copy the contents from the `bootstrap.scss` into your `_bootstrap-components` folder. You can then choose to comment out certain items (grids for example) if you opt not to use them.

Here's an example of how it might look like with Bootstrap.

```
1 // Bootstrap core mixins
2 // -----
3 • 3 @import "bootstrap-sass/lib/mixins";
4
5 // Bootstrap core css
6 // -----
7 // - When using Susy, bootstrap grid is not required
8 // - When not using code or tables, they are not required as well
9 • 9 @import "bootstrap-sass/lib/scaffolding";
10 •10 @import "bootstrap-sass/lib/type";
11 // @import "bootstrap-sass/lib/code";
12 // @import "bootstrap-sass/lib/grid";
13 // @import "bootstrap-sass/lib/tables";
14 •14 @import "bootstrap-sass/lib/forms";
15 •15 @import "bootstrap-sass/lib/buttons";
16
17 // Bootstrap core components
```

For every component that you choose to use, remember to add their corresponding JavaScript just before closing the `</body>` tag.

That sums up how to integrate Susy with any UI kit of your choice. I wrote about the exact steps you need to take to integrate Susy and Bootstrap in [this sitepoint article] if you are keen to find out more.

You might also want to check out the starter kits that integrate Susy with Bootstrap and Susy with Foundation in the full package.

A Quick Wrap Up

We have gone through some basics to integrating Susy with any UI Kit of your choice in this article. Each kit is written in an unique way and you would have to customize your approach in order to integrate them into your project.

Do take note that whenever you use a UI Kit or Framework, you are subscribing to how they define their HTML, classes and their site architecture, which may be different from how you want your HTML to be structured.

A better, though more painful, way is to create your own UI Kit. Give it your own variables, style it the way you usually do and fit them to your coding style.

Sass Architecture

CSS has always been a hot mess to deal with. When a project gets larger, we start to become afraid of changing CSS for fear of breaking other parts of the site.

How do we best code with Sass then? There are multiple opinions on how we can use frameworks like [SMACSS](#), [OOCSS](#) or [BEM](#) to organize how we code. All of them make sense.

There's no correct answer to this longstanding debate.

I'd like to share with you how I borrowed concepts from each of these different frameworks.

Let's start with BEM.

BEM

BEM is an acronym for block, element and modifier. It is a naming convention that uses double underscores `(__)` and double dashes `(--)` to create related classes.

Say we are creating a gallery that has 4 rows. This gallery by itself will be a block. We will have the following classes in both HTML and Sass.

```
<!-- html -->
<div class="gallery"></div>
```

```
.gallery {}
```

A gallery will have gallery items within it. We could also say that gallery items are children of the gallery element. Just like how we inherit our last names from our parents, these gallery items take the form of an element and inherit

the block name from their parents.

BEM calls these relationships elements, and we write them in the format

.block__element .

```
<!-- html -->
<div class="gallery">
  <div class="gallery__item"></div>
</div>
```

```
.gallery {}
.gallery__item {}
```

You may have siblings or relatives who share the same surname. In these cases, you would have some relation with them, but you definitely aren't identical.

Our gallery items may not be identical to each other either. One item may be more prominent than the rest because it's featured.

At other times, the gallery may be changed. It might have 4 items instead of 3 per row.

For these cases, we use the BEM modifier. They are used with the format

.block--modifier or .block__element--modifier . We will also use the same block or element classname with the modifier in the html.

```
<html>
  <div class="gallery gallery--4items">
    <div class="gallery__item gallery__item--featured"></div>
  </div>
</html>
```

Using modifiers this way has its perks, because we will be able to use the styles that are similar to the base item, and modify on top of it.

```
.gallery {  
  @include cf;  
}  
  
.gallery--4items {  
  // We don't need to write @include cf; any longer since the  
  HTML will container .gallery anyway  
}
```

That's BEM in a nutshell.

In my world, BEM is meant to be used for single components. Perfect for small projects, like a blog page.

However, I felt something was missing when I needed to reuse similar styles across different components.

Things started to get messy and there wasn't a way to quickly identify what styles were used across different components and what styles were meant for single components.

Even using the word components on both types sounds confusing! If we are using similar styles in different components, these styles could be abstracted into an object.

It's difficult to tell between components and objects in HTML or Sass without any extra help, which is why I started namespacing my elements after reading [Harry Roberts' excellent article about CSS namespacing](#)

Namespacing

Quick Trivia! How do you tell two friends with the same name apart if you cannot see them or hear them physically?

Well of course you can use last names!

The practice of namespacing elements is like giving last names to them. In the example I gave above, objects could be given the `o-` prefix as a namespace while components can be given a `c-` prefix.

Of course we don't just have two namespaces. Here are a couple of others borrowed from Inuit.css, SUIT, SMACSS and from Robert's article.

- `o-`: Object.
- `c-`: Component.
- `u-`: Utilities.
- `t-`: Theme.
- `l-`: Layout.
- `is-`: State indicator.
- `has-`: State indicator.
- `js-`: Javascript component.

Objects are abstracted styles that can be used across different components. They can be used in totally unrelated places. When we want to add or edit a style that propagates throughout the whole site, we use the object.

Components are isolated pieces that are styled independently from other components. This allows them to be placed anywhere in a website and they would still function as they need to. We will create components most of the time.

Utilities are extra classes that serve as overrides and do only one thing at a time. These classes are often given the `!important` suffix to override any existing styles on the element.

Themes are classes that are placed way above in the `<body>` to let us know if the site is using a certain theme, which we can then use a [theming mixin](#) to create multiple themes effortlessly.

Layout classes are used for laying out elements within the page. Most Susy code will fall into these category, as with Susy containers.

is- or **has-** are state modifiers that are only used if the object is in a certain state. Examples of these include `is-loading`, `has-loaded` and `is-open`.

js classes signify that JavaScript is targeting this class. It makes it incredibly clear that Javascript is involved in the component. I often use camel case for these classes in accordance to JavaScript convention.

When we namespace our elements, the HTML markup becomes extremely clear.

```
<div class="c-articles">
  <div class="c-article">
    <h2 class="c-article__title o-block__title"> ... </h2>
    <div class="c-article__entry"> ... </div>
  </div>
</div>
```

In this scenario the article title consists of two classes. An object,

`.o-block__title` that is used across components and a
`.c-article__title` that is created specifically for this component.

If a property has to be added to the article title, we can first determine if it needs to propagate throughout the site. If it does, we add it to the object. If not, we add it to the component.

This brings about a new level of clarity for both HTML and Sass. It does the same for our Sass architecture as well.

Sass Architecture

This is how I currently organize my Sass files, along with some examples of files that I used when creating a tutorial on replicating the layouts on 99u.com.

```
// Libraries
// -----
// For all libraries like Susy and Breakpoint
```

```
@import "lib/lib";

// Helpers
// -----
// For all functions and mixin helpers
@import "helpers/helpers";

// Variables
// -----
// For all variables used within the site
@import "variables/colors";
@import "variables/typography";
@import "variables/themes";
@import "variables/zindex";
@import "variables/breakpoints";

// Reset and base files
// -----
@import "base/reset";
@import "base/base";

// Layouts
// -----
// Susy and layouts go here
@import "layouts/susy";
@import "layouts/layouts";
@import "layouts/wrap-top";
@import "layouts/wrap-featured-videos";
@import "layouts/wrap-featured-secondary";
@import "layouts/wrap-footer";

// Objects
// -----
// Repeatable objects across the site
@import "objects/typography";
@import "objects/media";
@import "objects/buttons";
@import "objects/ads";
@import "objects/block";
```

```

// Components
// -----
// For individual, isolated components
@import "components/article";
@import "components/workbook";
@import "components/load-more-content";
@import "components/featured-videos";
@import "components/editor-picks";
@import "components/popular-posts";
@import "components/footer";

// Utilities
// -----
@import "utilities";

// Page Specific Overrides
// -----
// Rarely ever used. Exceptions may be homepage, for example

// Shame
// -----
// For new style snippets that haven't been sorted and commented
// yet
@import "shame";

```

Lib

The Lib folder (short for library) is a folder where I keep vendor libraries that I use with my Sass templates. Some examples of libraries I use are:

- Susy (Of course :))
- normalize-scss by Nicolas Gallagher
- adaptive-placeholder by Me
- mappy-breakpoints by Me
- vertical-rhythms-without-compass - Port of compass vertical rhythms, but without the use of compass by Me.

Helpers

The helpers folder (or sometimes named utils by developers), is where all the functions and mixins that help support Sass development comes in. This is where I place custom functions and mixins for the project.

Helpers are placed high up in the architecture because the mixins and functions must first be declared before they can be used.

Variables

Variables used within the project should be kept accessible, and hence isolated from the rest of the code.

Examples of variable files are:

- color: A map of all colors used within the project
- breakpoints: A map of all breakpoints used within the project
- themes: A map of themes and their styles for the project (Only used for projects with multiple themes)
- typography: A map of all typography elements used within the project. This includes font sizes, weights, variants and font-families.

Base

Base folders consist of common starter styles that I use for multiple projects. This allows me to scaffold projects quickly.

Since I prefer to start off my projects without any styles, I opt to remove margins and paddings from all heading and list elements

```
// Scss
h1, h2, h3, h4, h5, h6, ul, ol, p {
  margin: 0;
  padding: 0;
}
```

I'll also start every project with the `@establish-baseline` mixin to establish a vertical rhythm for the project.

Layout

Layouts is where I keep information related to layouts in a project. This is normally where all Susy mixins and functions reside.

Each new layout resides in its own file.

Objects

The object section holds all objects created for the project. Each object resides in its own file. Here are some examples of objects.

- block
- buttons
- media
- typography

As mentioned, any styles that propagate throughout the site can be abstracted into an object.

Components

This section holds all individual components created for the project. Each component resides in its own file.

Pages

Certain pages on a site can be very different from the rest of the pages. A prime example of this is the home page.

I try to create component-based styles and not rely on pages as much as possible. This section is empty most of the time.

Utilities

Utilities is a single file that consist of all utilities, which was explained earlier :)

Shame

Shame is a single file for you keep all the the Sass code that you have not had the time to sort through and document properly.

It is mainly a junkyard of code for a quick debugging session. This stylesheet needs to be cleaned out whenever possible.

Conclusion

A strong architectural foundation is the first step to having maintainable code. There's no correct way to write code. I encourage you to experiment and see what fits for you and your team.

Wrapping Up The Book

Thanks for taking the time to read through the entire book. You have come to the end of the Learn Susy book and I hope this book has helped to deepen your understanding of Susy, Sass and coding for responsive websites.

If you have any questions or just want to chat, I can be reached at zellwk@gmail.com. Drop me an email anytime, I'd love to chat :)

Special Thanks

Lastly, I want to express my heartfelt gratitude to these individuals who have helped me out tremendously along the way while creating this book. Without their support, I may never have been able to finish this book.

- [Eric Suzanne](#)
- [Sacha Grief](#)
- [Chen Hui Jing](#)
- [Scott Tolinski](#)