



Learning Susy

How to Make Ultra Flexible 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.

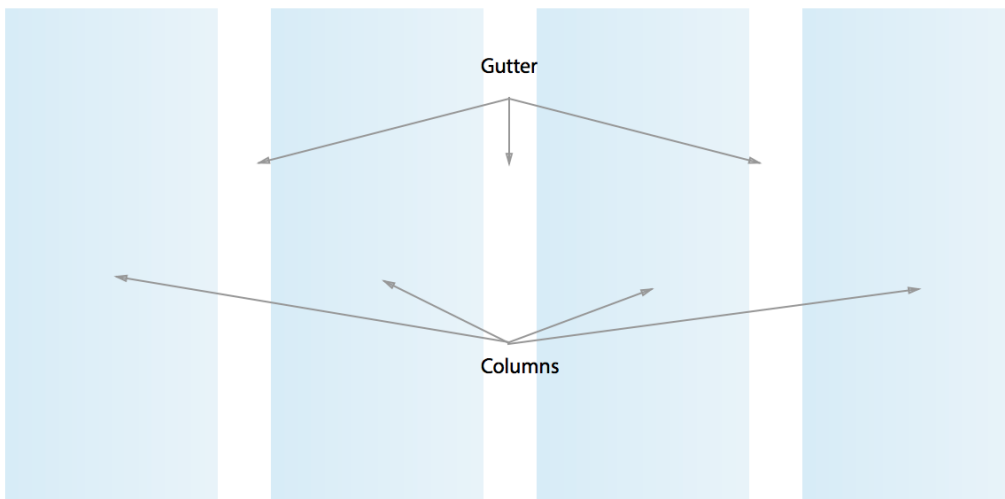
What is Susy?

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.

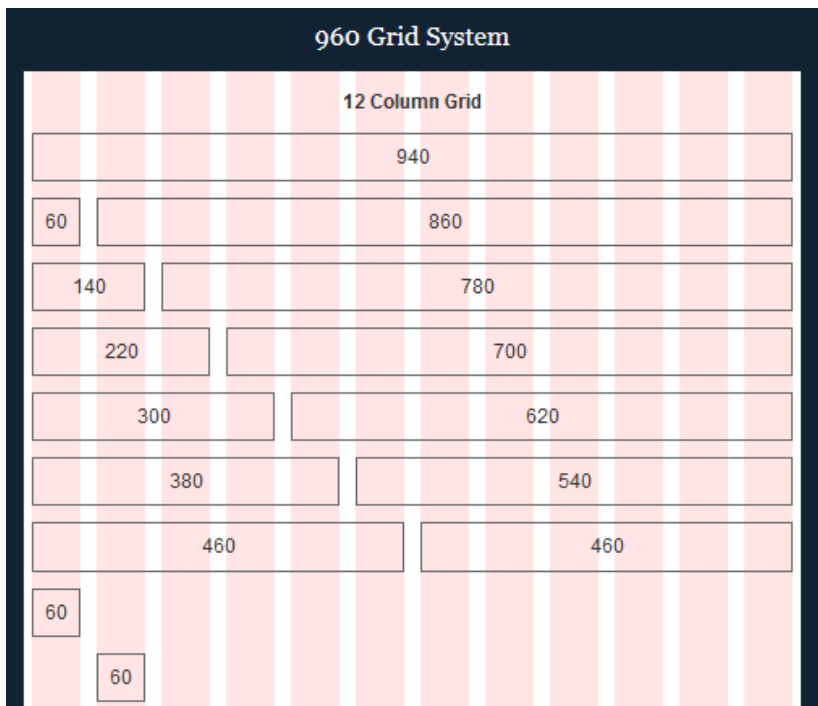


These vertical lines help to segment the page into two kinds of vertical spaces. We call the thicker space a column and the narrower space a gutter. The order and position of the elements on the web is known as a layout.

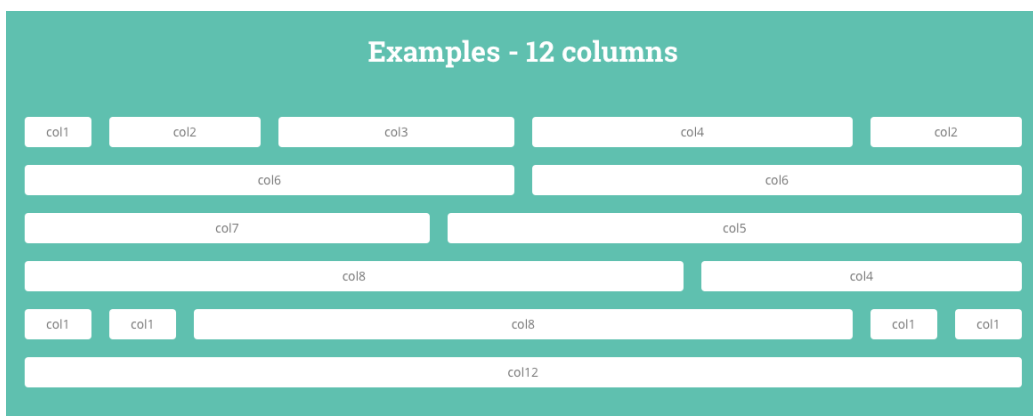


Grid systems have been around for quite a while, way before the web went mobile and responsive. You may even recognize some of these systems yourself!

The 960 grid system was one of the most widely used ones at one point. Every website used it as a based 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
```

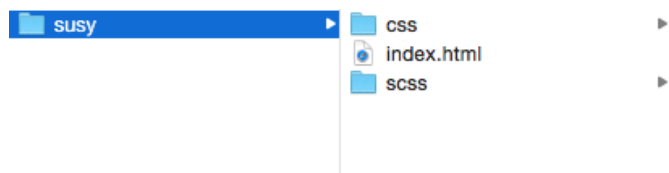
If you have previous versions of Sass and Susy installed, remove them with the `gem clean` command

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

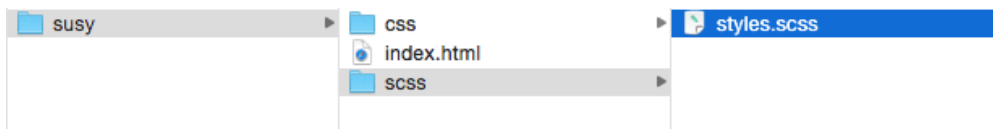
For Windows users, you must make sure you have [Ruby](#) installed on your system. You will also need to have Ruby Gems installed. Run the same code (without sudo) and you should have Susy and Sass installed.

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
```

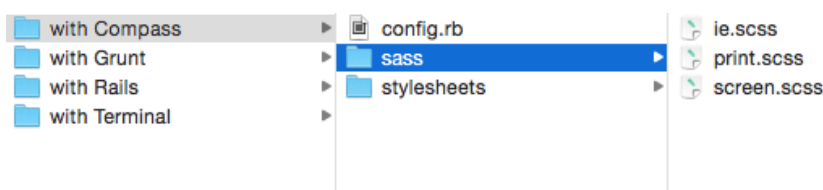
If you ran into any errors while compiling with Compass, I suggest you install the alpha version instead of the stable one. This would resolve most compilation errors.

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

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

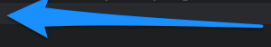
```
compass init
```

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'
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"
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.

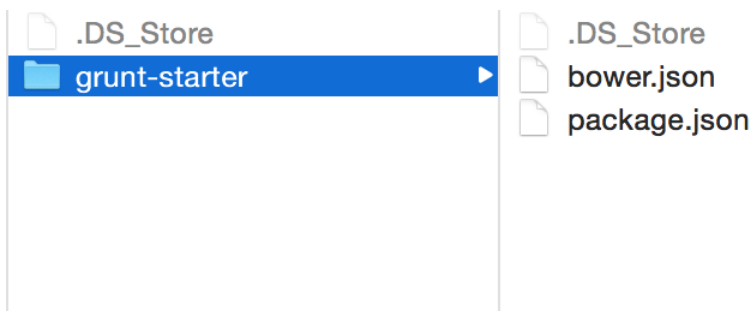
To do so, we require the `package.json` and `bower.json` files.

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')
    .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']);
});

```

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`.

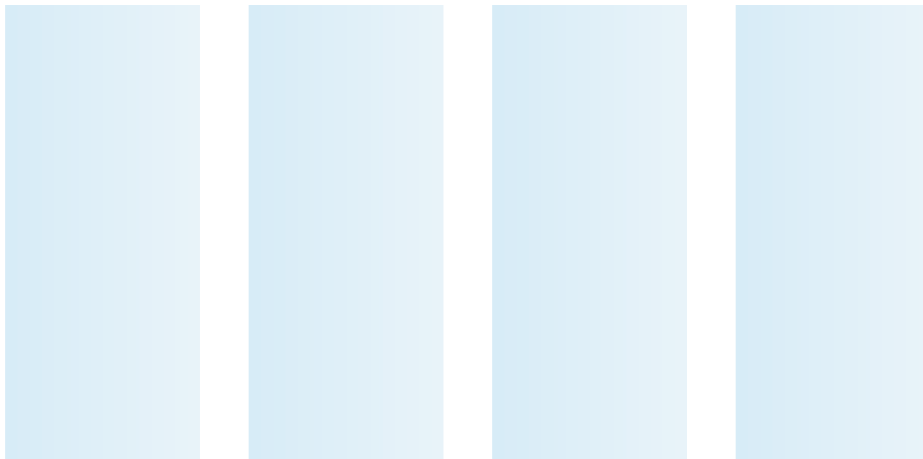
The good news is, you don’t have to know how to get the values out of the `$susy` map. You just have to know how to write them in.

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.

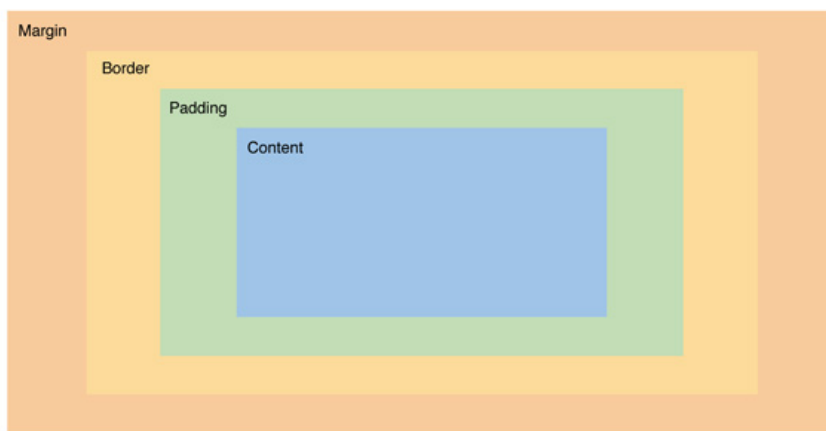
We have to tell the browsers to switch to the `border-box` sizing property by adding the `border-box-sizing()` mixin after the `$susy` map.

```
$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.

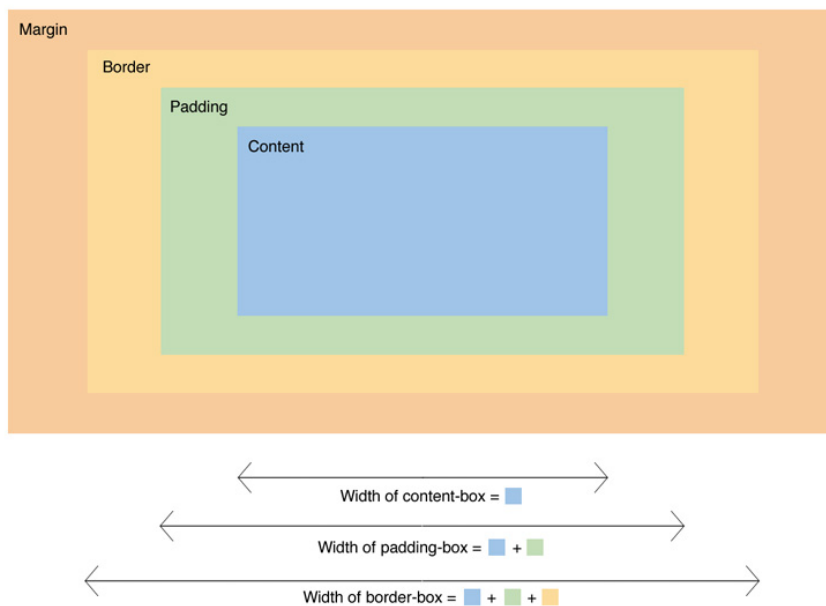


When there are this many layers, calculating the height and width of the HTML element becomes ambiguous. The `box-sizing` property is a cue for browsers to know what layers to include when measuring the height and width of the element.

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.

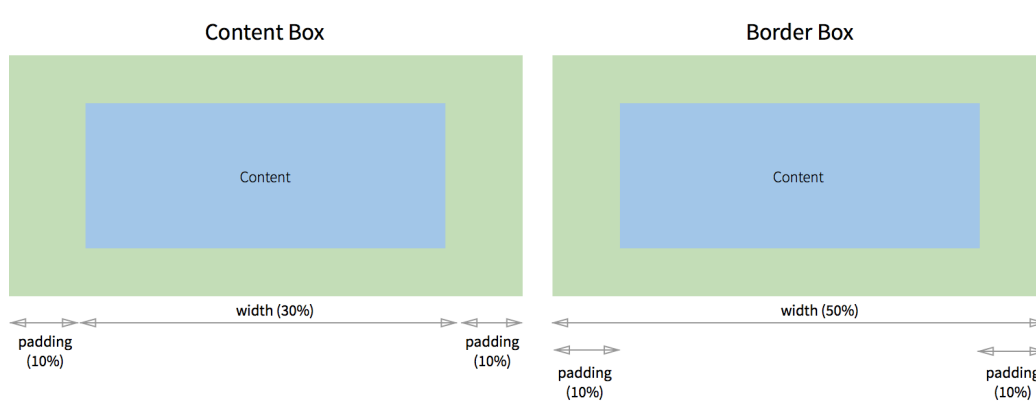
`padding-box` tells browsers that the CSS `width` or `height` property refers to the content section plus paddings. It's not recommended to use `padding-box` because browser support for it is not great.

`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;
}
```

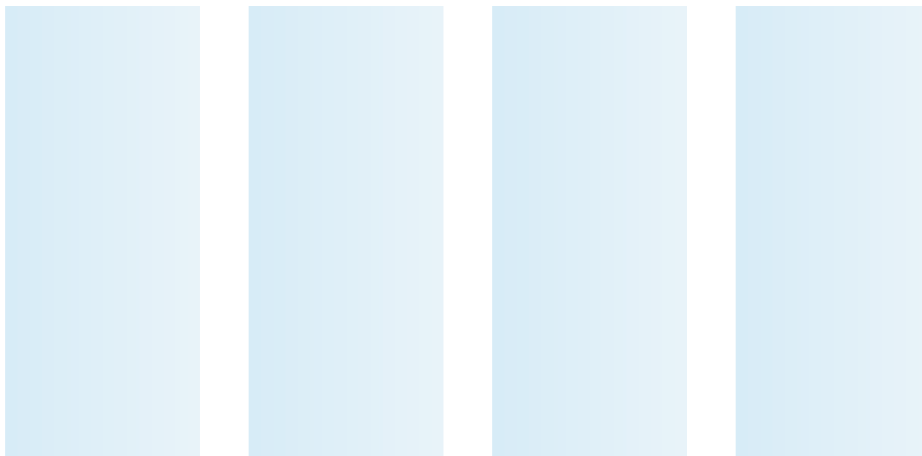
Using the Susy Map

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

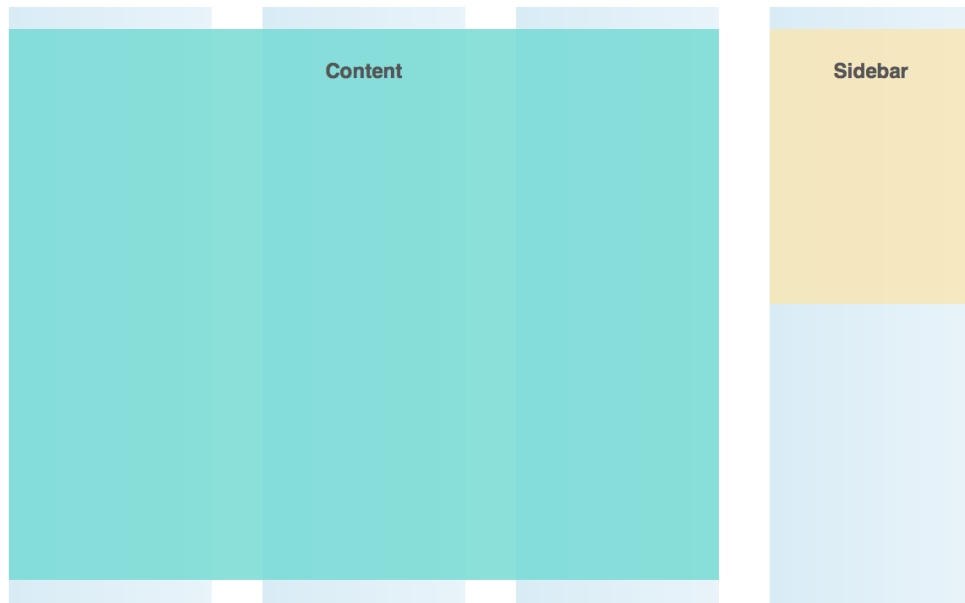
We have just covered the basics of Susy settings. You will need to know how to do this for every new grid that you create.

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

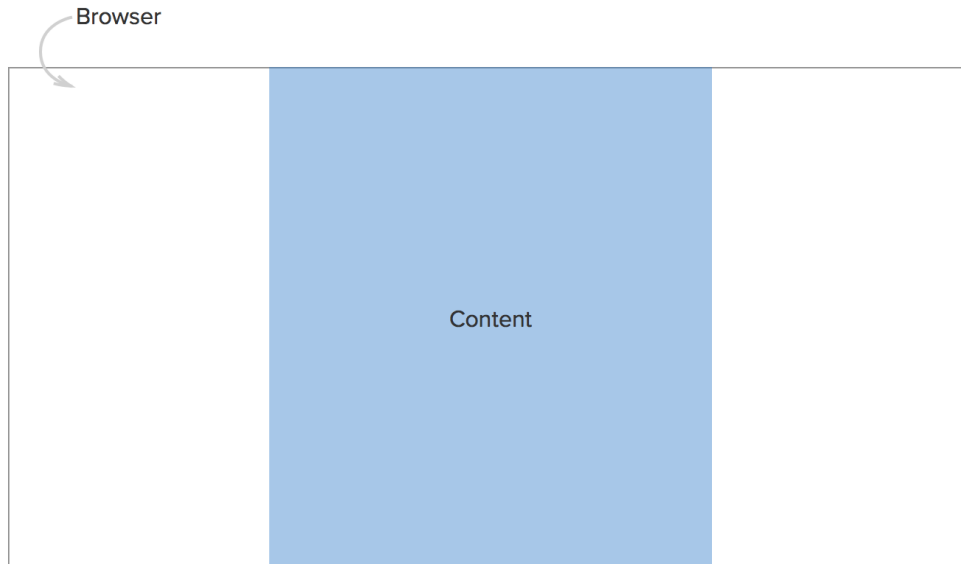
You will learn:

- 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.

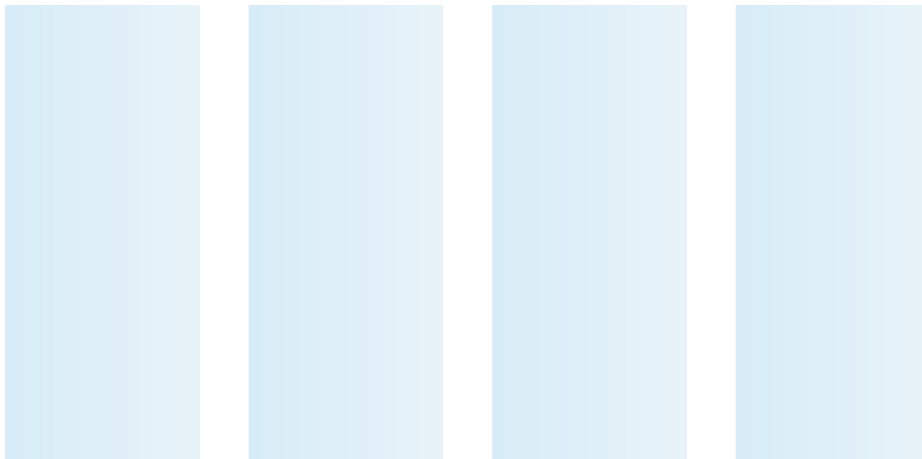
```
/* CSS */  
.wrap:after {  
  content: " ";  
  display: block;  
  clear: both;  
}
```

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>
```

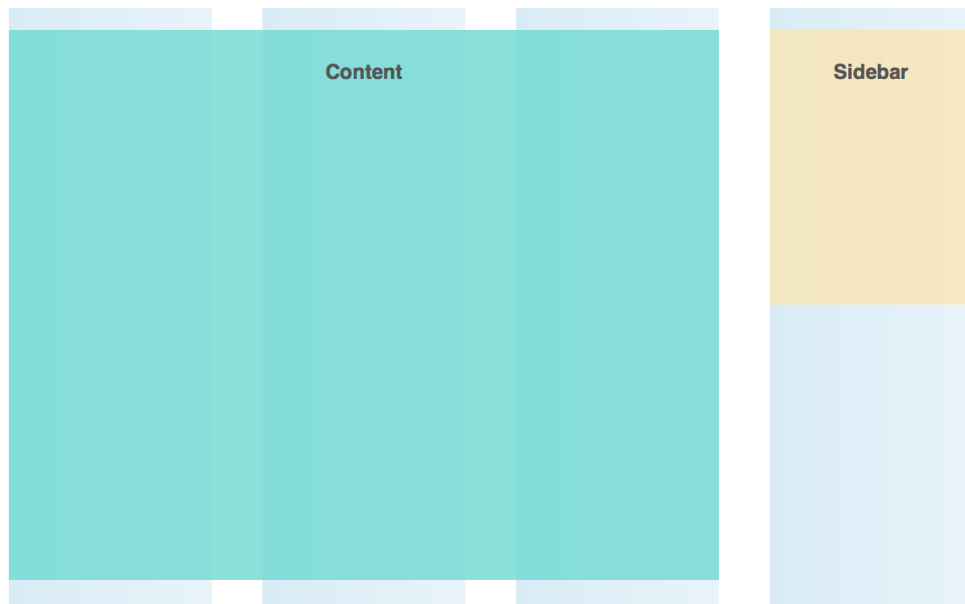
Classes used in this chapter are `.content` and `.sidebar` because it's easier to visualize and understand them. The `<h2>` tag is used within each `<div>` to act as the header element for the section. Some styles are also applied to `<h2>` so it looks solid.

[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.

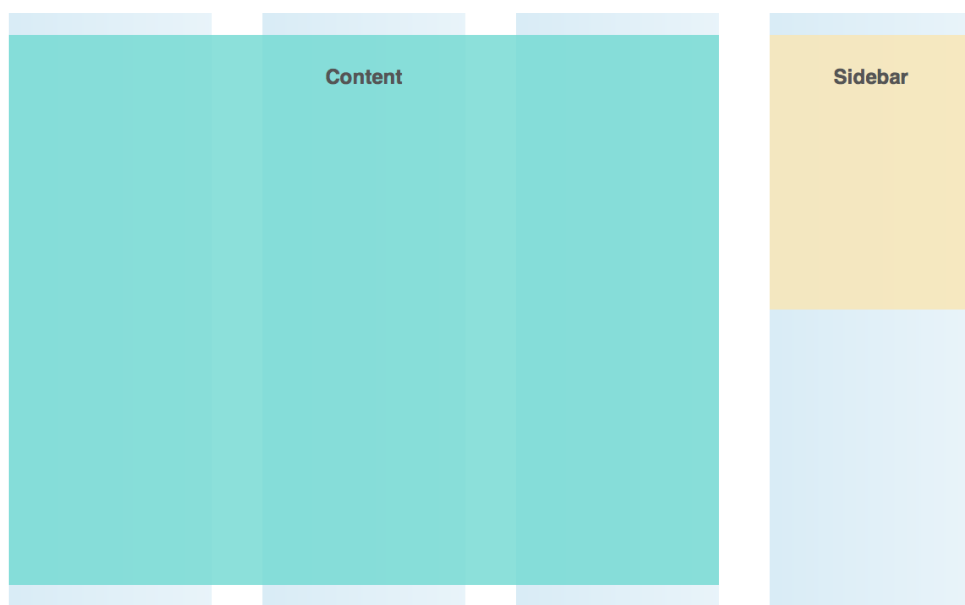
`$last` is an optional argument that tells Susy if this element is the last item in the row. The `last` keyword has to be supplied here for the last item on the row. We have to apply the `last` keyword for `.sidebar`.

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:

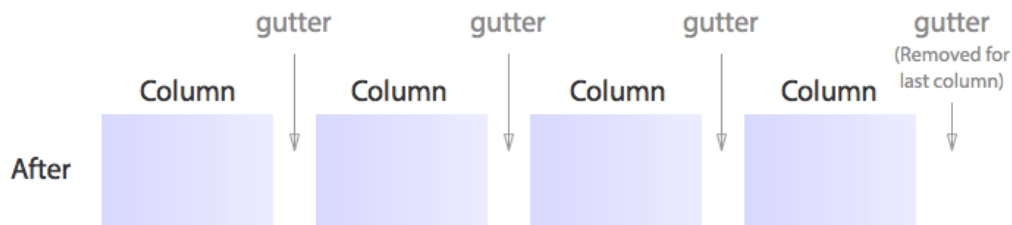
```
.content {  
  width: 73.68421%;  
  float: left;  
  margin-right: 5.26316%;  
}  
  
.sidebar {  
  width: 21.05263%;  
  float: right;  
}
```

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.

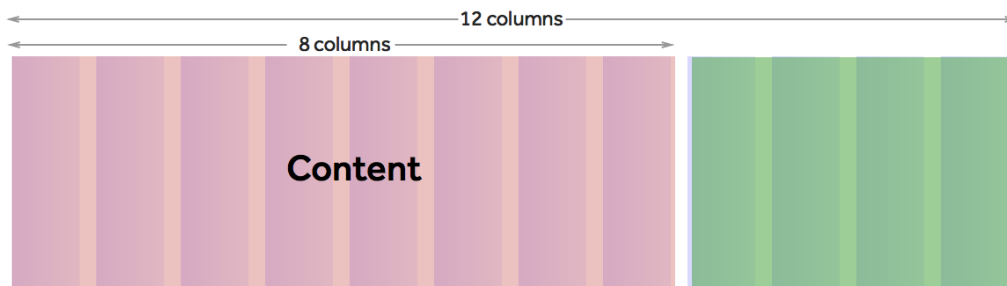
What is Context

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.

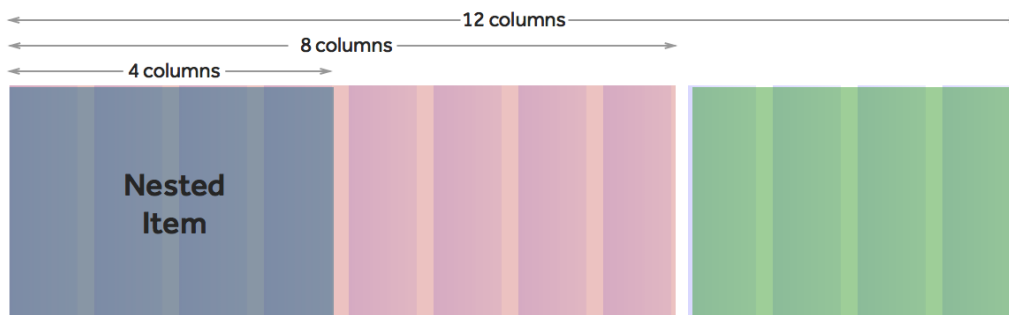
It can look something like this:

```
<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:

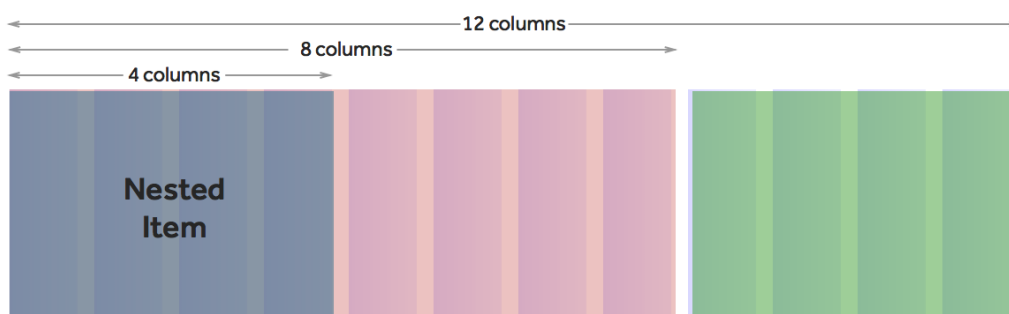


We know that `.wrap` is 12 columns, `.content` is 8 columns and `.inner-content` is 4 columns.

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);
}
```