

Up and Running with Sass

a practical introduction to
powerful CSS preprocessing

by Guy Routledge

AtoZ Sass: Up and Running with Sass

A practical introduction to CSS Preprocessing

Guy Routledge

© 2016 Guy Routledge

Contents

Introduction	1
What will this book cover?	1
Who is this book for?	2
What will you learn?	2
Chapter One: What is Sass?	1
What is a Preprocessor?	1
Meet the Superpowers	2
Sass Syntax	3
Two Syntaxes, Many Compilers	4
Chapter 2: Up and Running with Sass	6
Installing Sass	6
First Sass Project	8
Compiling Sass with Koala	13
Compilation Errors	18
Converting an Existing CSS Project to Sass	18
Chapter 3: Organising Sass with Partials	20
CSS Maintainability	20
Partials	21
Sass architecture	25
Chapter 4: Sass Fundamentals	27
Variables	27
Nesting	33
Mixins	38
Wrap Up	44
Next Steps	44
Never Stop Learning	45
Thank You For Reading	46
About the Author	47

Introduction

Sass is a CSS pre-processor. It's a tool that supercharges the way we write styles for web documents and it provides a powerful way to organise our styles, keep code modular, reduce repetition and enables us to work faster and smarter.

I've been making heavy use of Sass since 2012 and was introduced to it on my first day of joining the digital team at a top creative agency in London. I'd never used Sass before and was thrown in at the deep end on a huge project with a tight deadline. I needed to get up to speed with the basics *fast*.

After a bit of trial and error, I found my stride and I soon started to really enjoy the process of writing Sass - it's much faster and much more logical than "vanilla" CSS. Over the years I've dug deep into some of the more advanced features of Sass, hungry for knowledge and always looking to push the limits of what's possible and hone my skills.

In my day to day work as a contract front-end developer I've been exposed to many of the CSS preprocessors on the market including Sass, LESS, Stylus and postCSS. There are others available too, but this book (as you might have guessed from the title) is going to focus purely on Sass.

I may have used many of the CSS preprocessors out there, but Sass is my favourite by a long shot.

- The syntax feels more CSS-like
- When I first started using it it was by far the most feature-rich preprocessor
- There are a vast array of tools, frameworks and plugins that also use Sass
- And the Sass community is amazing, open, friendly and wonderfully quirky. I've attended local meetup events and flown around the world to attend Sass Conference.

But since you've already picked up a book about Sass I get the feeling that I don't need to convince you of its awesomeness. You're all in. You want to level up and sharpen your skills. You want to get up and running with Sass so let's get started.

What will this book cover?

This book will provide an introduction to Sass; what it is, where it came from, why it's useful, how to install it, how to use it, an overview of its features and details of the fundamentals: code organisation, nested selectors syntax, variables and mixins.

Reading and soaking up knowledge from the comfort of your sofa or office chair is one thing but nothing beats taking *action*. Throughout the book you'll find practical exercises, tips, best practices, links to further reading and actionable advice to get up to speed and firing on all cylinders as quickly as possible.

Who is this book for?

You're a web designer or web developer that's familiar with CSS but you've yet to get your feet wet with Sass. You get CSS, but are often frustrated at how repetitive it can be. You care about your work and you value your time and want to produce the highest quality projects. You want to keep things organised so that you, and your fellow team members, can collaborate painlessly. You want to level up your skills and accelerate your professional career.

If any of these statements are resonating with you, this book is perfect for you.

Perhaps you're more experienced. Perhaps you manage a team and are looking for a resource to recommend to existing staff and new hires to help them get up to speed quickly and efficiently on the technologies you use.

The material in this book assumes a working knowledge of CSS although being a CSS expert is definitely not a pre-requisite.

If you've already dabbled with Sass a little bit, there will still be value for you here as we'll cover a number of Sass best practices and dive into thy *why* as well as the *what*. This will solidify your existing knowledge and prime you for more advanced topics in the future.

What will you learn?

By the end of this compact introductory guide you will be able to:

- Start a new Sass project with confidence
- Convert an existing CSS project to Sass with ease
- Keep your code more organised and more modular
- Work faster by writing less code
- And use the most popular, professional CSS tool in the industry today

There's lots to cover so let's dive right in. I can't wait to show this stuff to you.

Let everyone else know you're levelling up and help spread the word about this book:

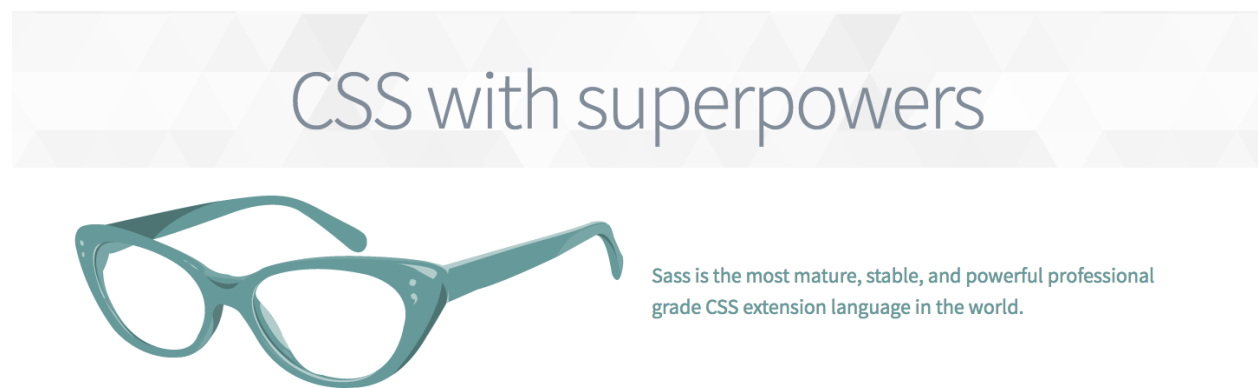
"I'm reading Up and Running with Sass by @guyroutledge and you can too.
So excited!"

[Click to Tweet this](#)

Chapter One: What is Sass?

Building things with HTML, CSS and JavaScript is so rewarding. Making stuff, playing around on the Internet all day and teaching everything I know is my dream job. My favourite part of what I do is turning a design into a reality by crafting things with CSS. CSS is great but sometimes it can get a bit repetitive and be a bit of a nightmare to maintain. Fortunately, Sass is here to save the day.

Sass is a tool that extends CSS with new functionality and the Sass team refer to it as “CSS with superpowers”. It doesn’t add any new visual features to CSS but is a game changer for the experience of writing the code itself. Sass makes the process of front-end design & development much more enjoyable and much more efficient, allowing you to write less code and have Sass do a lot of heavy lifting on your behalf.



Sass is the most mature, stable, and powerful professional grade CSS extension language in the world.

Screenshot from <http://sass-lang.com>

In this chapter we’ll start with a bit of background about what Sass is and where it came from. Next we’ll look at an overview of its features and the syntax. Finally we’ll wrap things up with a look at the different versions of Sass that exist today before moving on to getting set up and working through our first exercise.

What is a Preprocessor?

Sass is a preprocessor for CSS and it stands for Syntactically Awesome StyleSheets. It’s written “Sass”, not “SASS” and in case there’s any confusion, [@sturobson has you covered](#).

Preprocessors are available for lots of different languages. In front-end development we have preprocessors for HTML such as [Haml](#), [Slim](#) and [Jade](#). Javascript has its own

selection of templating libraries but also includes preprocessors such as [CoffeeScript](#), [LiveScript](#), [TypeScript](#) and [Babel](#).

Sass is not the only CSS pre-processor but it's certainly my favourite and the one I have used the most. If given the choice, I set it up with *every* new project I take on.

But what is a preprocessor? Well, here's a [technical definition from Wikipedia](#).

“a computer program that modifies data to conform with the input requirements of another program.”

When working with a CSS preprocessor we write a slightly different syntax to normal CSS but this is then “modified to conform with the input requirements” of a web browser. In short, the Sass is covered into normal CSS. This means we can *author* our styles one way but *output* them and package them up for a browser in a different form. By having this extra step in the process we can bring the power of programming to declarative languages like CSS.

So, to reiterate the point: instead of writing CSS, we write Sass which is then *compiled* into CSS. This CSS is then linked to the HTML document in the usual way.

```
<link rel="stylesheet" href="path/to/compiled/stylesheet.css">
```

Meet the Superpowers

The Sass that we write looks very similar to CSS (we'll take a look at the syntax very shortly) but running the Sass code through a compiler allows us to leverage all sorts of powerful features that are just not possible with CSS alone.

Here's an overview of some of the most popular Sass features. We can:

- Write code in **multiple files** that are then automatically combined together
- Automatically **compress code** to reduce file size
- Use **variables** throughout a project
- Reduce repetition by **nesting selectors**
- Reuse larger chunks of code with **mixins**
- Add logic like **conditional statements**
- Use additional flow controls like **loops**
- Create **custom functions**
- Manipulate colours with **colour functions**
- Perform **mathematical calculations** directly in the code
- Use **selector inheritance**

- and even more!

In this book we'll cover some of the fundamental features of working with multiple files, variables, nesting selectors and mixins.



For details on more advanced features, check out my free video series [AtoZ Sass](#).

Sass Syntax

Sass syntax comes in two flavours. The original syntax was whitespace dependent and, in contrast to the appearance of CSS, used no curly braces or semi-colons.

Code written in this style has a `.sass` file extension and looks a bit like this:

```
.box
width: 200px
height: 200px
background: #cc3f85
border: 1px solid #eee
```

While it takes a bit of getting used to, I actually quite like this syntax - especially if working in a Ruby or Rails project as it feels very “ruby-esque”. However, it’s by far the *least* popular of the two syntaxes and I don’t use it very often.

The second, more popular syntax looks identical to CSS - can you guess why it’s more popular? This syntax has a `.scss` file extension and when using this style of Sass, **any valid CSS code is also valid SCSS code** which makes the process of converting a project from CSS to SCSS quite painless.

Here’s some example SCSS code:

```
.box {
width: 100px;
height: 100px;
background: #cc3f85;
border: 1px solid #eee;
}
```

Throughout this book, I’ll be writing the `.scss` syntax as it’s the most commonly used and will look more familiar to you.

**Sass or SCSS?**

Sass is the name of the tool but it's also the file extension of the indented syntax. I've heard of the SCSS syntax being called "Sassy CSS" to better approximate its .scss file extension.

Despite the two very different syntaxes and different file extensions, when I talk about "Sass" I'm referring to the tool itself, not any particular syntax.

Two Syntaxes, Many Compilers

The original version of Sass was designed by Hampton Catlin (who's also the man behind [Hamli](#)) and was developed by Natalie Weizenbaum back in 2006. Since the initial version, Natalie Weizenbaum and Chris Eppstein have continued to extend Sass into the powerful tool it is today.

The original version of the language was written in Ruby but there are now a number of different compilers available written in different languages.

- [scssphp](#) is a Sass compiler written in PHP that only supports Sass up to version 3.2.12
- There's a [pure Java implementation](#) too
- [Libsass](#) is probably the best known port of the Sass compiler. It's written in C/C++ and has gained popularity for being blazing fast.
- [node-sass](#) is a library that provides bindings for Node.js to libsass.

Whether you choose the standard Ruby Sass or one of the alternatives above, the process of writing the Sass is identical. The major difference is the speed of compiling from Sass to CSS but there are also differences in feature support between the different compilers.

The Difference Between Ruby Sass and Libsass

Ruby Sass and Libsass are the two most popular versions of Sass in use today.

Ruby Sass came first and supports all the Sass features - because it's where all the features were first developed. However, Ruby Sass can be a bit slow to compile very large projects and running Ruby Sass requires Ruby to be installed in the development environment and often in the production environment too. Installing Ruby locally or on your server isn't always an option for everyone so it can be a bit of a stumbling block.

Libsass is a C/C++ port of the original Sass engine and doesn't rely on Ruby meaning the compiler can be implemented in other languages. Libsass doesn't actually do the compiling itself and requires a wrapper around it to run the library and compile your styles. There is a C wrapper called [SassC](#) and a JavaScript wrapper, [node-sass](#). The major benefit of using libSass is that it's much faster than Ruby Sass. It also provides a solution to the issue of Ruby being a dependency of Ruby Sass.

The features of Libsass are slightly further behind than Ruby Sass but are catching up steadily. Until Libsass and Ruby Sass reach feature parity, there will be no new features added to Ruby Sass. What a thoughtful bunch the Sass community are!



For details of just how fast Libsass is compared to other versions of Sass (and other preprocessors) check out this post on [Opinionated Programmer](#)

As we're dealing with the fundamentals of Sass in this book (rather than any of the more recent or more advanced features), all the examples will work in both Ruby Sass and Libsass. I personally have no experience of the PHP or Java versions so can't comment on what they're like to work with. Ruby Sass is still very popular and it will be our primary focus in the coming sections.

Chapter 2: Up and Running with Sass

In the previous chapter we learned what Sass is and little bit about its history and different syntaxes.

In this chapter, we'll get practical and install Sass, set up a local development environment and create our first Sassy project. Let's get to it.

Installing Sass

To use all the power that Sass has to offer, first we need to install it.

We'll start by installing Sass from the command line (on a Mac) and then take a look at some applications you can download and use if you prefer a graphical interface on Mac or PC.

Installing Sass via the Command Line

Using the command line can be daunting for beginners - I felt the same way my self when I first started. But it's a very useful skill to have and will open up a whole world of possibilities for improving your workflow and speeding up your development. Not only that, it's the quickest way to install Sass. If you'd rather not use the command line, the section beneath goes into details about various apps you can use instead.



Tip: Accessing the Command Line

Every Mac has an app called Terminal installed by default. For a super-fast way to open it and start commanding things, use Spotlight search. Hit the shortcut `cmd-space` to bring up Spotlight and start typing "terminal". Hit enter to open the app.

For a more powerful command line tool check out [iTerm2](#) and for a more powerful tool like Spotlight, check out [Alfred App](#).

We're going to install Ruby Sass which is done via Rubygems - a package manager for Ruby tools and libraries.

If you're on a Mac, Ruby will be installed by default but you may need to install Rubygems.



Details for installation can be found on the [RubyGems website](#).

With Ruby and Rubygems installed, we can now install Sass.

In your terminal, type in the following command. You can be in any directory as Sass is installed globally.

```
gem install sass
```

If you see an error about not having the right permissions then use the following:

```
sudo gem install sass
```

This will run the `gem` command with “super user” privileges. All being well, you should see a success message.

Congratulations, you’re done! Sorry if you were expecting more steps.

To double check that everything is working properly we can use the `sass` command to check the latest version.

```
sass -v
```

If you see output along the lines of `Sass 3.4.20 (Selective Steve)` (which is the latest version at the time of writing) then you’re good to go. We’ll look at how to use Sass now that it’s been installed in just a second.

Installing Sass with a GUI App

The command line is the quickest way to get up and running and work with Sass on a Mac. I’d encourage you to read the previous section to see just how few steps are required to install Sass via the terminal but I know it’s not everyone’s cup of tea.

So here’s a selection of GUI (Graphical User Interface) applications which will install Sass and act as your compiler. Many of these support other preprocessors too if you’re keen to experiment with other tools as well.

There’s [CodeKit](#) which is a full-featured paid product that does a lot more than just compile Sass files. I’ve not used it myself but I’ve heard very good things about it.

Other paid apps include [Ghost Lab](#), [LiveReload](#) and [Preprocs](#). Codekit is Mac only but the others are all cross-platform.

There's also a selection of free or open source solutions such as [Compass App](#), [Koala](#) and [Scout](#) which all work on Mac and PC.

When I teach beginner Sass classes in person, I encourage students to use [Koala](#) as its free and works on both Mac and PC. Its interface is very minimal which means it's quite quick to get used to. A little later we'll look at setting up a project in Koala and using it to compile your styles.

First Sass Project

Ok, we've installed Sass and are ready and raring to go. To take Sass for a test drive, let's start by setting up a bare-bones project and then look at how we write Sass and how it gets compiled into CSS.

For this exercise, I'll be using the command line but I'll explain the process which you can apply to a GUI "point and click" workflow if that's what you're more comfortable with.



Code Along: Compiling Sass

The best way to cement these skills is to put them into practice. To follow along with this exercise, create a folder on your desktop and open it up in your favourite text editor.

Create a Project Folder

So we have some files and folders to work with, create a folder for this exercise on your desktop called `sass-project`.

```
cd ~/Desktop
mkdir sass-project
```

Create Sub-folders for the Sass and Compiled Styles

Inside the `sass-project` folder create two sub-folders. One for your Sass code called `scss` and one folder for the compiled CSS called `css`.

```
cd sass-project
mkdir scss
mkdir css
```

**Tip: Naming things**

Always be consistent with how you name files and folders. I always keep all my names lowercase and use hyphens instead of spaces. This means I don't have to think about whether something was uppercase or lowercase or used spaces, hyphens or underscores when writing out file paths. If your team already use a particular convention, stick with it in the name of consistency. If your team doesn't have a naming process, suggest one!

Create a Sass File

In the `scss` folder create a new file to hold your Sass code. You may have a preferred name for your primary stylesheet (common ones include `style`, `main` or `global`) but I'm going to call mine `style.scss`.

```
cd scss
touch style.scss
```

Add Some Styles

In the `style.scss` file (your Sass file) add some styles in your text editor. You can add anything you like but below is some sample code for setting up some plain formatting for body copy and links. The styles you use are not important for this exercise to work but should be valid CSS.

Sample code for style.scss

```
body {
  color: #333;
  font-size: 100%;
  font-family: 'Avenir', 'Arial', sans-serif;
}
a {
  color: #f00;
  text-decoration: none;
}
```

**Remember**

Any valid CSS code is also valid Sass code in the SCSS syntax

Compile Sass to CSS

Right, we have a file containing some Sass code.

Back in the terminal, we can now compile our Sass to CSS. This may feel somewhat redundant here as our Sass is *already* valid CSS, but bear with me, we're illustrating a point.

Ensure you are in the `sass-project` folder and then run the following command:

```
sass scss/style.scss:css/style.css
```

If the compile was successful you won't see any message telling you so. If you do see a message, it will be an error message. Go back to the `style.scss` file and ensure you have no syntax errors.

The command above uses the `sass` command to compile our Sass to CSS. The compiler will run once and compile the input file `scss/style.scss` to the output file `css/style.css`. The input path is to our existing Sass file and the output path is where the CSS will be generated to. The input path and output path are separated by a `:` colon character.

It would be a bit tedious to have to run this command every time we made a change to our Sass and would really slow down our process. Sass is supposed to make things *faster* not slower!

To make Sass compile automatically we need to add an option to the command.

```
sass scss/style.scss:css/style.css --watch
```

Using the `--watch` flag means Sass will watch our files for changes and recompile whenever we save. When you start Sass with the watch flag you should see the message

```
>>> Sass is watching for changes. Press Ctrl-C to stop.
```

Go and make a change to your `style.scss` file and you should see a message in the terminal saying the a change was detected and the compiler has run again.

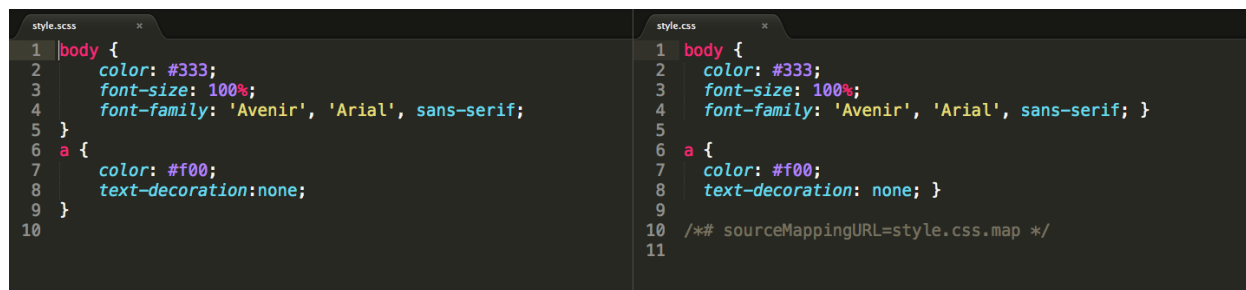
```
>>> Change detected to: scss/style.scss
```

To stop Sass just hit the keyboard shortcut `ctrl-c`.

The Compiled CSS

Even though we haven't used any fancy Sass features, the compiler still runs and converts the code in the `style.scss` file into CSS in the `style.css` file. From now on, any updates or additions to the styles of a project should be done in the Sass files, not the CSS files.

Here's a screenshot of the two files side by side:



Sass on the left, compiled CSS on the right

Look closely at the two files. All the selectors, properties and values are identical but the output format is slightly different in the generated CSS file; the closing brace of each style block is on the same line as the final declaration.



Source maps

The compiled stylesheet also contains a comment at the end about source mapping. This is a special data file which provides info that links the compiled CSS to the original Sass code for debugging purposes.

Control the Output Style

The difference in the output style above can be controlled when running the compiler. There are a few options available:

- Nested
- Expanded
- Compact
- Compressed

To use one of these options we pass another flag to the `sass` command:


```
sass scss/style.scss:css/style.css --watch --style=compressed
```

Nested

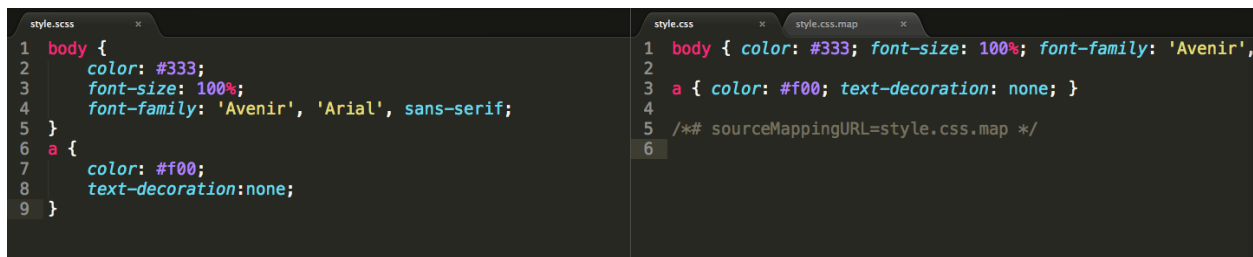
The nested style is the default and it places the closing brace on the last line of the style block.

Expanded

The expanded style is the same as the original, classic CSS style of having each property and brace on their own line.

Compact

The compact style is analogous to single line CSS where the selector and all the properties are on a single line.



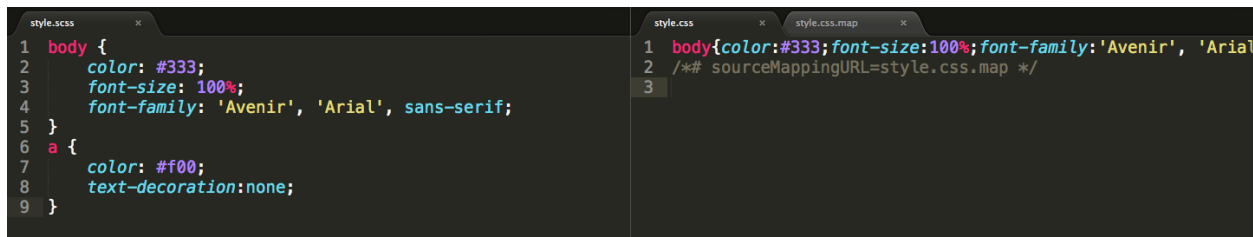
```
style.scss
1 body {
2   color: #333;
3   font-size: 100%;
4   font-family: 'Avenir', 'Arial', sans-serif;
5 }
6 a {
7   color: #f00;
8   text-decoration: none;
9 }

style.css
1 body { color: #333; font-size: 100%; font-family: 'Avenir',
2
3 a { color: #f00; text-decoration: none; }
4
5 /*# sourceMappingURL=style.css.map */
6
```

Compact output style

Compressed

The compressed output style is like minified CSS. Everything is on a single line and all the whitespace is removed.



```
style.scss
1 body {
2   color: #333;
3   font-size: 100%;
4   font-family: 'Avenir', 'Arial', sans-serif;
5 }
6 a {
7   color: #f00;
8   text-decoration: none;
9 }

style.css
1 body{color:#333;font-size:100%;font-family:'Avenir','Arial
2 /*# sourceMappingURL=style.css.map */
3
```

Compressed output style

This compressed style would be a nightmare to read as a CSS author, but the machines don't care. In fact, this is the most efficient way to send CSS code to the browser as it's the smallest possible file size.

This is the whole point of preprocessors: we write our code in a nice, developer-friendly format but the resulting CSS is only ever read by the browser so can be as compact and compressed as possible. Packaging up the CSS like this has performance benefits and should encourage other team members to leave the CSS file alone.

This is important so I'll reiterate: **when working with Sass you *never* edit the CSS files; only edit the Sass files.**

**Tip: When to use output styles**

During development just use the default output style so that the code is easier to read and debug. Code in production (your live environment) should be compressed or minified for performance benefits. This minification could be done locally and then uploaded to the live server or, ideally, run as part of an automated process on the server itself. Having automated processes for things like minification reduces the risk of someone forgetting (or not bothering) to compress the code before deployment.

When using the output style option, a new file named `config.rb` will be added to your `scss` folder. This is a configuration file which saves information about the output style being used. It can also be configured with details of paths to common folders for things like images, HTML templates and JavaScript.

Compiling Sass with Koala

We've used the power of the command line to successfully compile our first Sass project. But if you prefer a GUI workflow, here's how to do the same thing with [Koala](#), a free, cross-platform app for working with preprocessors.

Create a project folder

Create a new project folder with the following structure and a `style.scss` file in the `scss` folder.

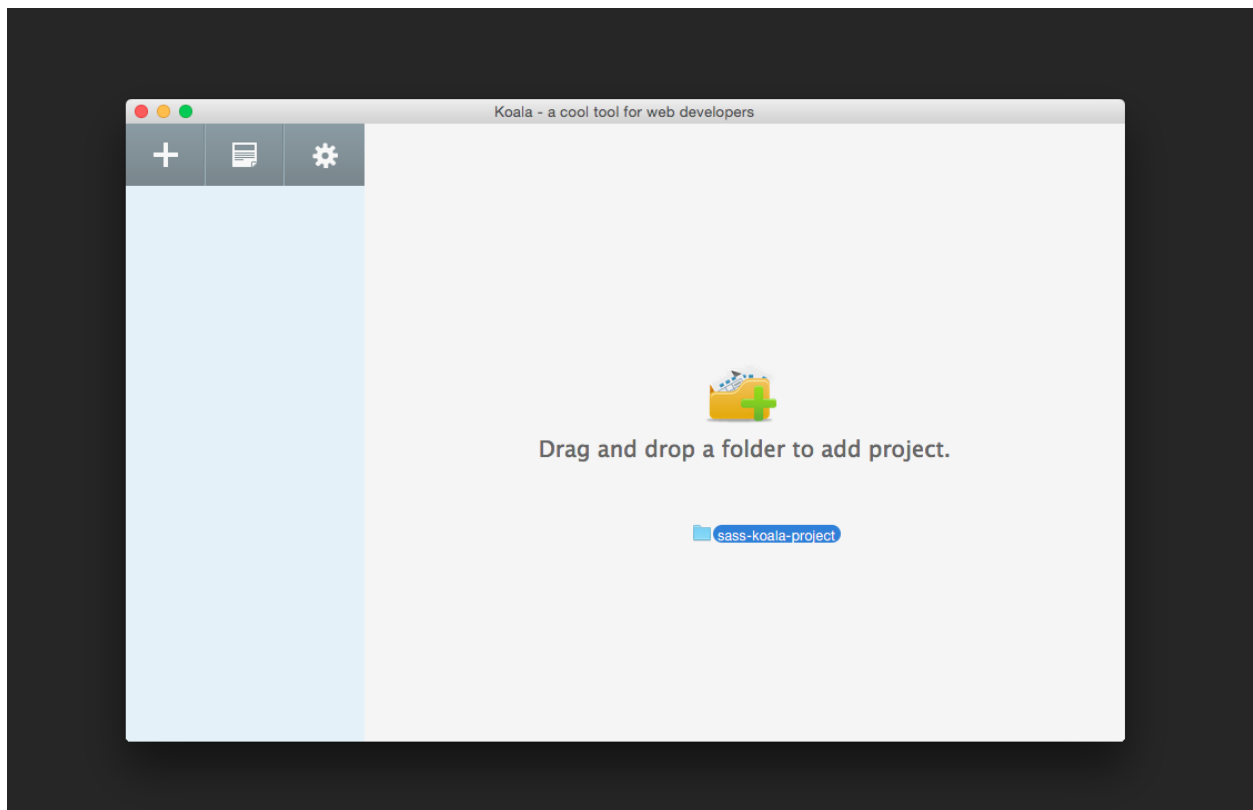
```
| - sass-koala-project
  | - scss
    | - style.scss
  | - css
```

Add some plain CSS to `style.scss` so you have something to compile (we'll add some *real* Sass features soon, I promise!).

Sample code for style.scss

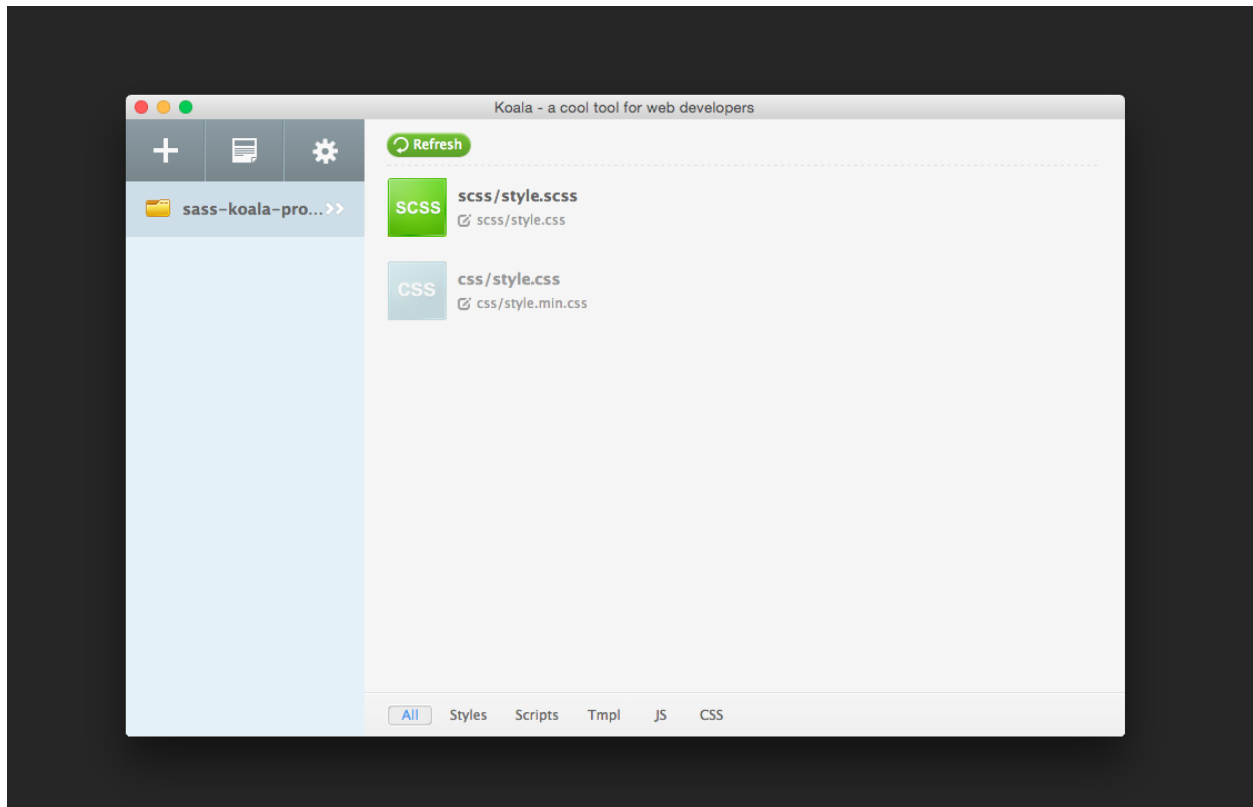
```
body {  
  color: #333;  
  font-size: 100%;  
  font-family: 'Avenir', 'Arial', sans-serif;  
}  
a {  
  color: #f00;  
  text-decoration: none;  
}
```

Launch the Koala app and drag and drop the `sass-koala-project` project folder on to it to create a new project in Koala.



Drag and drop to create Koala project

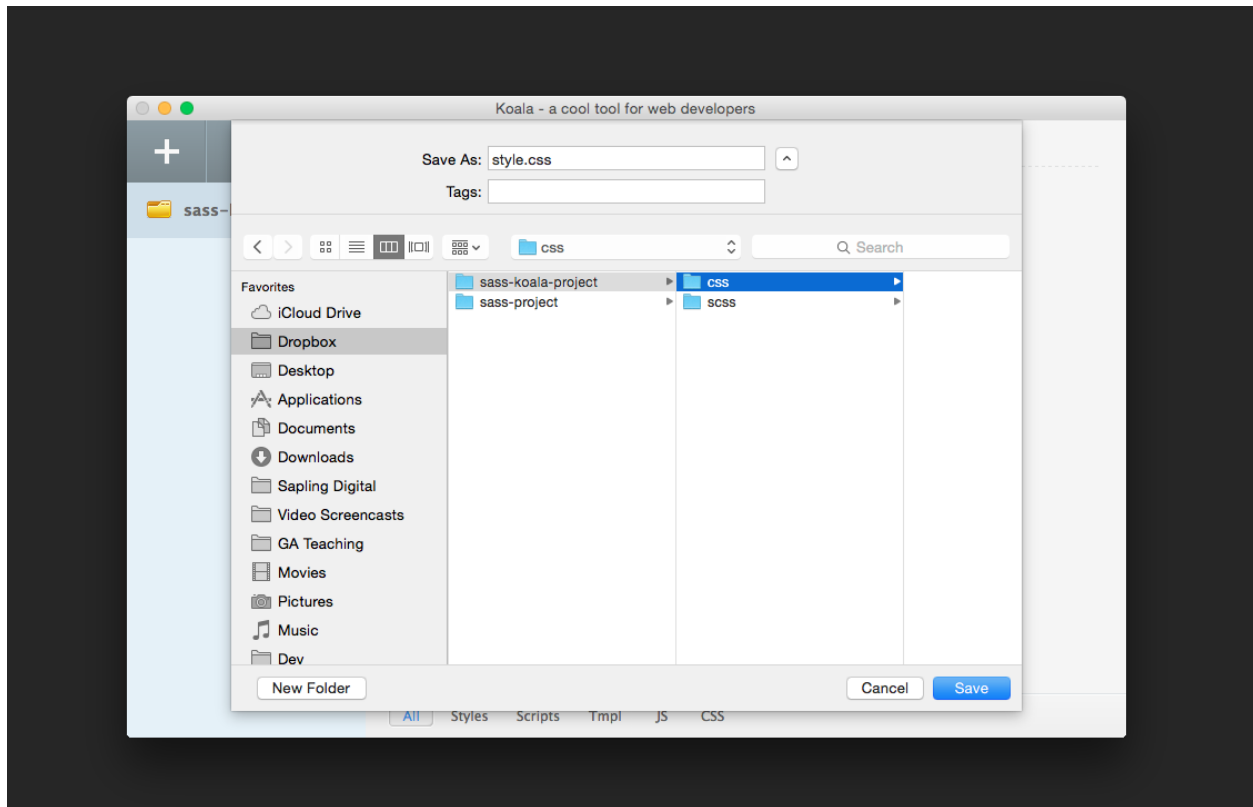
Koala will recognise the Sass file and try to be helpful by setting the default compile path. However, it may guess incorrectly. If Koala detects a CSS file, you can right-click and Remove that.



Koala detects file types it can work with

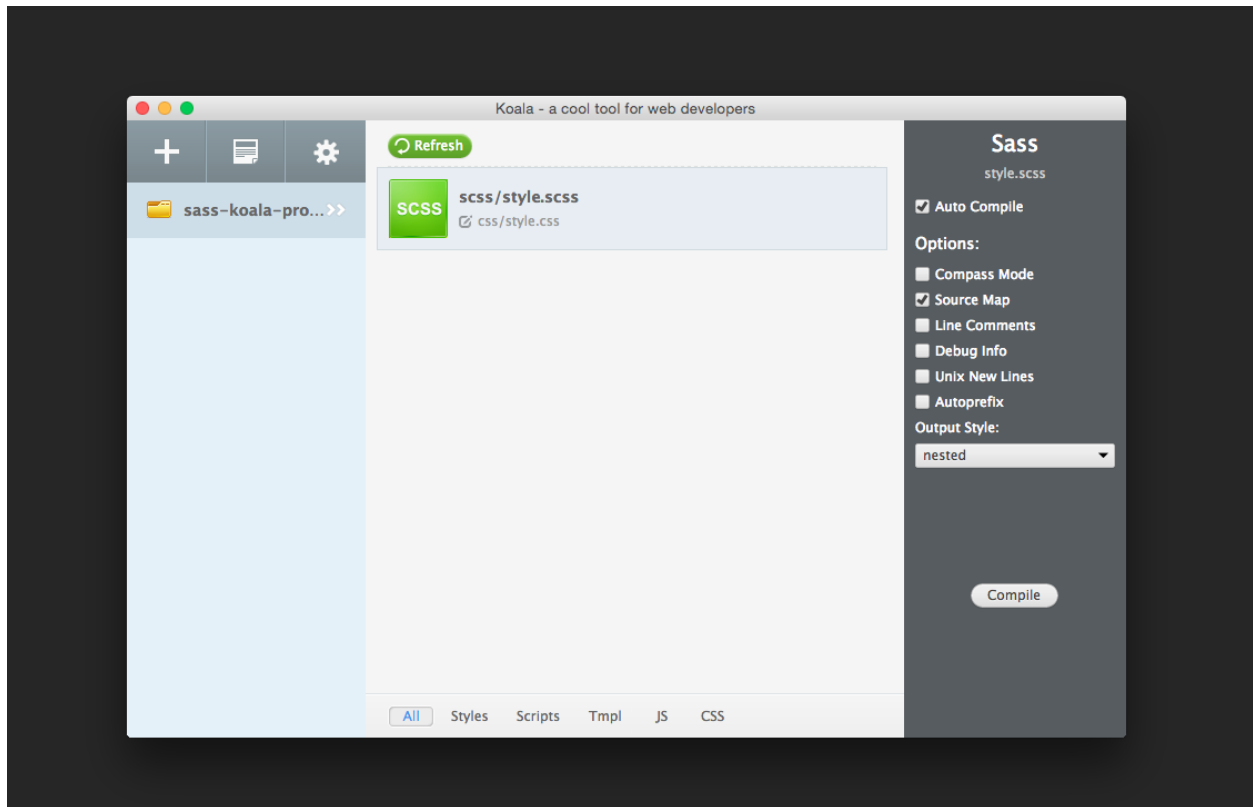
To ensure the Sass is compiled to the desired location, click on the edit icon for the SCSS file type to set the output path for the `scss`.

From the file browser that pops up, select the `css` folder and specify the filename `style.css`.



Set the output path to css/style.css

Click on the large green `scss` icon and a sidebar will fly out from the right. This is where you can configure options about how the Sass is compiled. Ensure the "Auto Compile" box is checked.



Drag and Drop to Create Project

Now click the “Compile” button to run the compiler for the first time. All being well, you should see a success message. Now each time you save the `style.scss` file, Koala should automatically re-compile the Sass.

Open the project in a text editor and check out the `css/style.css` file. You should see your styles looking almost identical to the code you added to `style.scss`.



For details of the difference in output style, see the previous section.

Koala has a few other settings that can be configured through the sidebar in the app. I'd encourage you to have a play around with them to see what they each do.

The most powerful option is the “Autoprefixer” option which will run your compiled CSS through [Autoprefixer](#) and automatically add any vendor prefixes that your code should have for cross-browser compatibility. This means you don't have to ever write a vendor prefix ever again!



Are you a command line user and want the added bonus of this automatic prefixing too? I bet you do. I'll be producing a video for AtoZ Sass about autoprefixing and using other command line tools like [Grunt](#) and [Gulp](#) for compiling your styles. [Sign up for the mailing list](#) to ensure you don't miss out.

Compilation Errors

Sometimes we have errors in our code. It happens to the best of us and it's not something to panic about even though it can be frustrating.

When writing normal CSS, errors like a missing semi-colon or a mistyped property or value can go unnoticed if we're not looking in just the right place at just the right time. CSS fails silently (some say gracefully) and any invalid declaration is just skipped over. If there's an error in a selector, that whole selector is ignored.

When writing Sass, if there's a syntax error in the Sass code, the compile won't be able to finish compiling and an error will be thrown. This will be displayed in a big red box if you're using Koala or appear as a message in your terminal:

```
>>> Change detected to: scss/style.scss  
error scss/style.scss (Line 7: Invalid CSS after "color: #f00": expected "{", was ";")
```

Not only will Sass alert you to the error, it will also point to the line number where the error occurred. This can really help with debugging and will help you get back on track quickly. When the error is fixed, Sass will recompile and patiently wait for the next save.

This is a much more useful behaviour than CSS which just ignores stuff it doesn't understand and will greatly reduce the risk of small mistakes going unnoticed.

So, even though we've not technically written any *real* Sass yet, we've already seen some great benefits of using it, including compressing the generated CSS and notifying us of any errors in our code as we write.

Converting an Existing CSS Project to Sass

Learning stuff is fun and rewarding, but nothing beats putting your new found skills into action on a real project.

Not everyone starts a new project every week so sometimes it can be hard to start using new techniques or technologies as you learn them. Even if the opportunity does

arise, it can be daunting to use a new tool for the first time or it may be a struggle to get everyone onboard with your desire to experiment.

But the great thing about Sass is that any valid `.css` file is also a valid `.scss` file. This makes transitioning from a CSS project to a Sass project very straightforward. To convert a CSS project into a Sass project there are a just a handful of steps:

1. First, make sure everyone is on board if this is a “real” project - especially if it involves anyone other than yourself.
2. Rename your current `css` folder to `scss`.
3. Rename all your current `.css` files to `.scss`.
4. Set up the project with your preferred compiler (command line or an app like Koala).
5. Start writing Sass.

To practice your new found skills and increase your confidence in using Sass on a real project, here’s a practical exercise.

**Exercise: Convert an existing project**

Find an existing project of your own and make a copy of it. Follow the above steps to convert it from a CSS project to a Sass project. You can use this as a playground for experimenting with the Sass features we’ll be discussing over the next few chapters.

Not only will this exercise put your learning into action, but having “before Sass” and “after Sass” versions of a project can be great collateral in convincing your boss or your fellow team mates about making the jump to Sass in the future. Good luck!

Chapter 3: Organising Sass with Partial

Having installed Sass and setup our environment in the previous chapter, we're now ready to start our discovery of the magical features of Sass.

In this chapter we'll look at one simple Sass feature which goes a long way to help organise and plan the front-end architecture of a site. We'll look at the number one issue with vanilla CSS projects, how we can solve that with Sass and look at some best practices for structuring the files and folders in your future projects.

CSS Maintainability

Writing CSS is fun! But the most painful thing about CSS on a real-world project is maintaining it. That's not fun. It's complex and difficult.

Writing CSS is often a journey of discovery; building out the various components or pages of a project, experimenting with effects, adding and removing styles as you go, re-ordering sections, and iterating as the best way to complete the work becomes clear. Sometimes all the designs are available up front, sometimes the project scope change half way through. But building websites is a creative process and things can always be tweaked and changed. As a project grows, the good intentions of the CSS authors can become muddled by last minute updates, browser hacks and even the most reasonable client requests.

Maintaining a CSS project is hard but it's even harder and more complex when working with a team of developers rather than a single person. You've probably worked on a project with tens of thousands of lines of code in the CSS file. You've probably found yourself searching through such a file thinking "where should I put these updates". You may even find that a legacy project with a messy CSS file contains some pretty crazy selectors and a good sprinkling of `!important` declarations too.

A complex, untamed stylesheet is often not due to lazy or inexperienced developers (although that multiplies the problem) but it's often the result of poor maintenance and poor organisation of the project from the start. When all (or most) of the styling information for a project gets bound up in a single file, it's not a huge surprise when things go from bad to worse.

Having one long CSS file makes it difficult to find the right place to make changes or add new code. Often the solution is to just add new styles at the end of the file but this can sometimes cause specificity issues.

Another more dangerous situation occurs when multiple people need to work on the single stylesheet at the same time. What happens when they both add styles at the end of the file? What happens if they reorganise whole sections of code? What happens if they both change the same line of code?

If the project is version controlled, any conflicts will be caught but they will need to be resolved (sometimes manually) which is not *always* a straightforward process and can be fraught with human error. If the project isn't version controlled, things can very quickly unravel and the likelihood of code conflicting or being overwritten is dangerously high.

Personally, I wouldn't *ever* want to work on a project that didn't use a CSS preprocessor or version control ever again.

In many cases, a single file stylesheet is good for performance (ie. there's only one HTTP request for the styles) but it's terrible for maintenance. Perhaps there's a better way? Perhaps Sass can help us out? Spoiler alert: yes it can.

Because Sass is run through a compiler, we can author styles in multiple files and then bring them all back together at compile time (a process known as concatenation). This allows separate pages or distinct components to have their own Sass file which is much better for organisation and hugely reduces the issue of conflicts or work being overwritten.

Partials

In the previous chapter we used the command line `sass` tool which compiled a file named `style.scss` into `style.css`.

There's a convention in Sass that any file with an `.scss` file extension will be compiled into a corresponding `.css` file with the same name. For example:

```
style.scss    -> style.css
products.scss -> products.css
admin.scss    -> admin.css
```

But with Sass we can create multiple *partial* files and combine them into one that will be compiled using the Sass `@import` statement.



CSS `@import` vs. Sass `@import`

CSS and Sass both have an `@import` statement. The CSS one makes an HTTP request for each imported file whereas Sass doesn't make any requests. The Sass `@import` is used by the compiler to combine all the imported partials into one file. That one file is then linked to the HTML document in the usual way and a single request is made by the browser.

To differentiate the partial files from the file(s) that will be compiled, we name them starting with an `_` underscore character.

From the previous chapter, we had a sample project with the following structure:

```
| - css
  | -- style.css
| - scss
  | - style.scss
```

You can see this structure in your terminal if you install the `tree` command. You can do this via [homebrew](#) - a package manager for OSX.

```
brew install tree
```

If you prefer to use the file Finder or aren't on a Mac, you can view the folder structure using your standard file explorer.

To demonstrate how Sass partials work, let's work through the following exercise:



Exercise: Using Sass Partials

Work with the Sass project from the previous chapter or create a new folder with the same structure as above to follow along with these steps.

Navigate to the project folder

Inside the project folder, we're going to create a new Sass partial file called `_base.scss` in the `scss` folder.

Open the project folder in a text editor and ensure your preferred Sass compiler is watching for changes.



Sensible Default Styles

In every project I tend to have a `_base.scss` partial where I put all my "sensible default" styles; things like default heading and link colours, consistent margins and padding for form elements and list items, that kind of thing. It's a bit like a CSS reset like [normalize.css](#) but these base styles are more specific to the design of the project rather than just to make elements more consistent across different browsers. In fact, my `_base.scss` is a customised version of `normalize.css` which I keep updated from project to project.

Add Some Base Styles

So we have some styles to compile, add the following to the newly created `_base.scss`. You can use whatever CSS you like but I'm using this to illustrate the kind of styles I might add as my sensible defaults.

Styles for `_base.scss`

```
body {  
  font-size:100%;  
  font-family: Helvetica, Arial, sans-serif;  
  background:#f1f1f1;  
}  
h1, h2, h3, h4, h5, h6, p, li, input {  
  margin-bottom:1rem;  
}  
a {  
  color:#cc3f85;  
}  
a:hover {  
  color:#b23430;  
}
```

If you save this file and take a look at your compiler, you'll notice that Sass hasn't detected any changes and the compiler hasn't run. If you open the compiled `style.css` file, you won't see any of the styles that have just been saved in `_base.scss`.

Import the Styles

To make use of these styles, they need to be "imported" into `style.scss`.

To import the partial into the main stylesheet, we use the `@import` statement followed by the name of the partial *without* the underscore and without the file extension.

Open the `style.scss` file and scroll to the top. Add the following line:

```
@import 'base';
```

Save the file and keep an eye on the compiler. Sass will detect the change and the CSS will be compiled. If you now take a look in the compiled `style.css` file, you should see the base styles appear first, followed by the rest of the contents of `style.scss`.

**Tip: Where should @imports go?**

It's possible to `@import` a file anywhere within any Sass file (including partials and Sass files that will be compiled). I always declare all of my imports at the top of the file so it's clear what partials are available in the current stylesheet. I don't often import partials into other partials as it can sometimes get confusing to follow the trail of import statements to find the file I'm looking for.

This method of splitting up styles into their own file goes a long way to solve the maintainability problems mentioned earlier.

Partials in the Real World

We've just learned how multiple files can be combined into a single stylesheet using partials. But this is quite a short example and may not provide enough clarity on how partials would be used in a "real" project.

In a real project, it's common to have many partials split over multiple folders (we'll look at a folder structure next). It's common to have one compiled stylesheet that's used on all pages in the project and perhaps a handful of other stylesheets for complex sections of a site or app that may not be seen by every user. This could include a dashboard stylesheet for logged in users or a stylesheet for the eCommerce process for an online shop.

Each compiled stylesheet, should only contain `@import` statements. As such, a primary compiled stylesheet may look as follows:

Importing partials into style.scss

```
@import 'variables';
@import 'utilities';
@import 'base';
@import 'header';
@import 'footer';
@import 'sidebar';
@import 'home-page';
@import 'about-page';
@import 'contact-page';
@import 'blog';
@import 'products';
```

The specific pages would be different between projects but I'm sure you get the idea: each distinct component or page in the project has its own Sass partial. This separates concerns, keeps code organised and adds meaning to the project structure which

improves understanding for other team members and makes maintenance much more manageable.

Sass architecture

Splitting up chunks of code into separate files is all well and good but for medium to large scale projects, sometimes even more planning and more advanced structuring of styles is required.

CSS architecture is a hotly debated topic and much of it also comes down to personal preference. Over the years I've developed a system that works for me and I'll outline it here for you. If you're working with an existing team then it's likely they have their own way of structuring Sass projects and it would be wise to find that out first. If you normally work on your own, you may have more flexibility in experimenting with different project architecture to see what works best for you.

Firstly, I keep all my assets (css, Sass, scripts, images etc.) in an `assets` folder. My `scss` folder contains two sub-folders: `includes` and `global`. My primary Sass stylesheet which is loaded on every page is called `global.scss` because it gets used everywhere. This goes in the `scss` folder. Any additional compiled stylesheets also live in the `scss` folder, have their own folder of partials and will also reference partials from `includes`.

This structure can be visualised as follows:

```
| - project root
  | - assets
    | - css
    | - fonts
    | - images
    | - js
    | - scss
      | - includes
      | - global
      | - products
      | - global.scss
      | - products.scss
```

Includes is a folder of Sass partials that generate little or no CSS when compiled (we'll discuss this in more detail in the coming chapters).

The `global` folder (and any other folder for compiled stylesheets) is further divided into four sub-folders.

- Layout

- Components
- Pages
- Vendor

These folders all contain additional Sass partials.

```
| - scss
  | - includes
  | - global
  | - layout
  | - components
  | - pages
  | - vendor
```

The `layout` folder contains styles for the big-picture layout elements. Things that are present on multiple (or every) page of a site like the site header, site footer, or sidebar.

The `components` folder contains a series of partials for each distinct component of the project. Each file has a single concern. Typically I'll have a partial just for button styles, one for form styles, one for social icons, user avatars, modal dialogs etc. Every component on the site is split out into its own file and as far as possible, components are built flexibly and should be able to be used anywhere on the site without causing weird side effects.

The `pages` folder is for page specific styles these are typically quite short files as the vast majority of layout and UI is handled by the base styles and components.

Finally, the `vendor` folder is used for library code from other sources. This might include styles for plugins, frameworks, grid systems or open source libraries of utilities from 3rd parties.

Every project is different but this folder structure has served me well across a range of different types of projects from mobile prototypes to custom Wordpress themes to Ruby on Rails applications.



Exercise: Reverse engineering your favourite website

Take a look at a recent project you've designed or built. If you can't think of one then use a popular site you visit a lot. Start with the homepage and dissect the various visual elements into different categories of layout, components and pages. Make a note of them and write them down in each category. This will get you into the habit of thinking modularly rather than seeing a design as whole pages full of content and hundreds of different styles.

In the next chapter we'll combine this knowledge of partials with other Sass features and start working towards some more significant examples and even more maintainable and modular code. I can't wait!

Chapter 4: Sass Fundamentals

Although we've yet to really play with anything visually interesting or mind-blowing, I hope you're getting excited by the prospect of bringing all these powerful Sass features to your next project.

Now we're getting to the really interesting stuff; the core parts of the language that will really supercharge your ability to style web pages like a pro. You can go incredibly deep with all sorts of advanced Sass features or you can just pick and choose a couple of things that you find useful. Regardless of which path you choose, you'll want to have a solid understanding of the fundamentals. So in this chapter we'll be going into detail on the three of the most important Sass features:

- Variables
- Nesting Selectors
- and Mixins

Variables

When I first started using Sass, the feature that instantly sold me was variables. From a maintainability and efficiency point of view, this has saved me hundreds of hours of development time.

Variables are a feature of all programming languages and are a way of storing a value (or series of values) for use throughout the execution of a program. Variables provide more meaning to abstract values as we can give them an easy to remember, human-readable name and then reference them by that name whenever we want to use them. The value of a variable can change (or vary) throughout their lifetime so they are a very flexible way of passing around data. CSS isn't a programming language, it's a declarative language, but Sass brings some of that programatic feel and the addition of variables to our toolbox is incredibly useful.



Native CSS Variables

Native variables (or more accurately "custom properties") are on their way into the CSS spec. However, it will be a long time before browser support is deep enough to use them confidently. In the meantime, preprocessing offers a great alternative.



For more info about native CSS variables, check out this excellent article by Philip Walton: [Why I'm Excited About Native CSS Variables](#)

Working with Sass Variables

To declare a variable in Sass we prefix the name of the variable with a \$ dollar sign. The value assigned to that variable is specified after a : colon character.

A classic example is to create a variable for the primary or brand colour of a website so that it can easily be re-used multiple times.

```
$brand-color: #cc3f85;
```

The variable can then be used by referencing its name.

```
a {  
  color: $brand-color;  
}
```



Watch out: Hyphens or Underscores

Sass variables tend to be declared in hyphenated case as hyphen-separated words is the norm in CSS. However, hyphenated variable names can interchangeably be written in underscore case. If you declare \$brand-color you can use that variable by calling \$brand_color (and vice versa). I'd advise against this arbitrary swapping of hyphens and underscores because it's confusing. Pick one style and be consistent.

When the Sass is compiled, any reference to \$brand-color is substituted for whatever is stored in the variable and the end result is plain ole vanilla CSS.

Compiled CSS output

```
a {  
  color: #cc3f85;  
}
```

Variables are incredibly useful and when it comes to working with colours, saves the time and hassle of constantly looking up hex codes or running the risk of incorrectly using a colour-picker tool and ending up with 100 shades of brand colour scattered throughout a project.

But not only are they useful, colour variables (and other types too) convey so much more meaning than a hex code. Do you know what colour `#cc3f85` is? Probably not (it's AtoZ CSS Pink, by the way). But if I asked "what is the brand colour of AtoZ CSS?" it's a much more meaningful question.

Variables make code more readable and reduce repetition but they can also save developers masses of time.

I was once working on a project where the client wanted to make a slight adjustment to their brand colour throughout a huge site. They had tons of pages, loads of blog posts, products, recipes, competitions and games, and a lot of CSS. I was asked if it would be a big job to make the update throughout the site and test everything to make sure we didn't miss anything. They were quite surprised when I said

"I'll do it right now"

And ten seconds later had updated a single line of Sass and shown them the change across the entire site on my screen right in front of them. That's the power of Sass; a change that affected over 100 pages was made by editing a single line of code. Thanks, Sass. You made me look smart!

Variable Types

Sass can handle all sorts of different types of variables including:

Colours

We've already seen use of hex colours, but Sass can handle any of the other valid colour formats as well.

```
$primary-color: #cc3f85;
$secondary-color: orange;
$tertiary-color: hsl(200, 50, 50);
$hover-color: rgba($secondary-color, 0.5)
```

Lists

List variables can be used for things like font-stacks or shorthand CSS properties. They can be space-separated or comma-separated and you can even have a list of variables!

```
$font-stack: 'Avenir', 'Helvetica', 'Arial', sans-serif;
$default-margin: 0 0 1em 0;
$heading-font: italic 20px/30px $font-stack;
$font-sizes: $font-size-h1, $font-size-h2, $font-size-h3
```

Strings

Strings of text (text surrounded by single or double quotation marks) for things like urls or pseudo element content

```
$background-image: '../..img/body-bg.jpg';
$content: 'lorem ipsum';
$fallback-font: 'Times New Roman';
```

Numbers and length values

Numbers can be integers (whole numbers) or floats (decimal numbers). Numbers with a unit such as px, em, % etc. are known as length values.

```
$magic-number: 42;
$global-line-height: 1.4;
$padding: 20px;
$margin: 1em;
```



Tip: Maths with length values

If you multiply a number by a length value, the result will take the same unit as the length value. Eg. $2 * 1.5em = 3em$.

If you divide a length value by another length value with the same unit, the result will be a number without any unit. Eg. $42px / 2px = 21$.

Boolean values

Boolean values are `true` or `false`. These aren't that commonly used but come in handy when combined with more advanced features like conditional statements, loops or for authors building frameworks or component libraries.

```
$output-media-queries: true;
$hide-for-mobile: false;
```

And finally, a new variable type added in Sass 3.3, is map variables. This is a named store of key and value pairs. These act a bit like JavaScript objects or Ruby hashes and are very powerful for storing multiple named values in a single variable.

```
$colors: (  
  red: #f00,  
  pink: #cc3f85,  
  dark-blue: #045b9d,  
);  
$z-layers: (  
  modal: 9000,  
  modal-bg: 8000,  
  header: 6000,  
  background: 0,  
  behind: -1  
);
```

There's lots to talk about regarding maps, and lots of Sass functions for setting and getting the values stored within. There's a whole video for [AtoZ Sass](#) about maps: what they are, how they work and some examples where they're really, really useful.

Variable Naming Conventions

There are lots of different types of information we can save in Sass variables and they can each be used in many different ways. We can use variables to provide more meaning to our code and help our fellow developers understand our intentions.

But due to the many different types of data that they can hold and the ease in which they can be used, it's good to develop a system for naming variables in a meaningful way to reduce the risk of confusing our team rather than helping them out.

Take the following example of a registration form and some initial Sass styles.

```
<form class="form-subscribe">  
  <div class="form-field">  
    <label for="name">Name:</label>  
    <input type="text" id="name">  
  </div>  
  <div class="form-field">  
    <label for="email">Email:</label>  
    <input type="text" id="email">  
  </div>  
  <input type="submit" value="Subscribe">  
</form>
```

```
$border-color: #ccc;

.form-subscribe {
  padding: 20px;
}
.form-subscribe input[type="text"],
.form-subscribe input[type="email"] {
  padding: 20px;
  border: 1px solid $border-color;
}
```

The value 20px has been used in a couple of places to define consistent padding within the form and the form inputs.

We could create a variable called \$padding that stores the value 20px and replace all instances of that number.

```
$border-color: #ccc;
$padding: 20px;

.form-subscribe {
  padding: $padding;
}
.form-subscribe input[type="text"],
.form-subscribe input[type="email"] {
  padding: $padding;
  border: 1px solid $border-color;
}
```

Now let's say we want to add a bit of vertical spacing between each .form-field and want to use the same 20px to keep all the spacing consistent.

We could add margin: 20px or could use the variable as it already contains the value we're interested in. But then we get something that doesn't really make sense:

```
.form-field {
  margin-bottom: $padding;
}
```

Instead, it would be more flexible to refactor this and use a more generic variable name like \$global-spacing. Now we can use this value for margin, padding, border, or any other property that takes a length value. We've used the power of variables without writing confusing or misleading code. Win win.

I like to take this idea of naming conventions further and have developed a system of variable prefixes that I use throughout a project.

- `$color`- for colours
- `$font`- for anything font related like families, sizes, line-heights
- `$media`- for named media query breakpoints
- `$image`- for anything to do with images, such as paths or dimensions
- `$global`- for any global values like spacing

Having a system like this goes a long way to making the code more readable and helps with the age old programmer's dilemma of "naming things is hard".

When working with variables they're often needed throughout a project and will be used in multiple files. So it's clear what variables are available throughout a project, it's a good idea to create a `_variables.scss` file which can then be `@imported` into any compiled stylesheet as necessary.

There's more to variables but this is a great starting point. In the final section of this chapter we'll take a look at another Sass feature which allows reuse entire blocks of code rather than just a single variable value. But first, we've got another popular Sass feature to learn about.

Nesting

Another firm favourite feature of preprocessors is the ability to nest selectors. To understand what nesting refers to, let's first take a look at some normal CSS code. Here's some hypothetical styles for setting up the layout of a site's main navigation.

```
.site-header {  
  position:relative;  
}  
.site-header .site-nav {  
  float:right;  
}  
.site-header .site-nav ul {  
  margin:0;  
  padding:0;  
}  
.site-header .site-nav li {  
  display:inline-block;  
}
```

With the exception of the first selector (`.site-header`) all the other selectors are *descendent* selectors; classes within classes or elements within classes. While there is nothing wrong with this way of writing CSS, the repetition of `.site-header` and `.site-nav` on multiple occasions can sometimes be a little tedious.

Nested selectors

Using the nesting feature of Sass, the above snippet could be rewritten as follows.

```
1  .site-header {  
2    position:relative;  
3  
4    .site-nav {  
5      float:right;  
6  
7      ul {  
8        margin:0;  
9        padding:0;  
10     }  
11     li {  
12       display:inline-block;  
13     }  
14   }  
15 }
```

So what's different?

Well, the major difference is the indentation of the code and the reduced repetition in each selector.

On line 1 the `.site-header` class is defined and its opening curly brace appears at the end of the line. The closing curly brace for this selector is found on line 15. The remaining selectors are *nested* (indented) within the braces of the `.site-header` selector.

On line 4 the `.site-nav` class is defined. Because this has been nested within the `.site-header` class, the compiled CSS is

Compiled CSS

```
.site-header .site-nav { }
```

On line 7 we see a `ul` selector. Because this is nested within the `.site-nav` class, which is nested inside the `.site-header` class, the compiled CSS for this selector would be

Compiled CSS

```
.site-header .site-nav ul { }
```

This nested syntax brings more order and meaning to the hierarchy of the styles - much like we use indentation in HTML code to show the parent, child and sibling relationship of elements in the document.

**Tip: DRY**

DRY stands for Don't Repeat Yourself. It's a common mantra in programming to ensure code is lean rather than unnecessarily bloated. The repetition in the class names in the previous example is necessary for specificity so is perhaps as DRY as it can be from a technical perspective. However, the nesting feature of Sass allows us to author code with less repetition but still ensure the correct selectors are generated in the compiled code.

Nesting selectors can bring more meaning to the code and reduce the amount you have to type but **care should be taken not to nest too deeply** - more on that in a minute.

Nesting pseudo selectors

Pseudo selectors like pseudo elements or pseudo classes are recognised by the presence of a : colon character in the selector. A hover state is a classic example:

```
button {  
  color: #000;  
  background: #fff;  
}  
button:hover {  
  color: #000;  
  background: #fff;  
}
```

Using our knowledge of nesting from above, we may attempt to achieve this with the following Sass code:


```
button {  
  color: #000;  
  background: #fff;  
  
  button:hover {  
    color: #000;  
    background: #fff;  
  }  
}
```

But this won't work. The compiled output will be:

Compiled CSS

```
button { color: #000; background: #fff; }  
button button:hover { color: #fff; background: #000 }
```

We still want to use nesting for pseudo selectors but we need a way of joining the `:hover` part to the parent selector. We can achieve this with the `&` ampersand character which has a special meaning in Sass.

```
button {  
  color: #000;  
  background: #fff;  
  
  &:hover {  
    color: #000;  
    background: #fff;  
  }  
}
```

The `&` ampersand character inserts the parent selector in the current nesting hierarchy - think of it a bit like a special variable which always stores the value of the parent selector. The compiled CSS is now correct:

Compiled CSS

```
button { color: #000; background: #fff; }  
button:hover { color: #fff; background: #000 }
```

The Sass ampersand is very powerful and has a few more tricks up its sleeve but this should suffice as an introduction to its core purpose and most common use case.

Deep Nesting

Being familiar with HTML indentation should make Sass nesting quite an intuitive practice but as with anything new it will take some getting used to. But be careful. Don't over-nest your selectors as this will result in CSS with very high **specificity** that won't be reusable across your project.

```

5 |         color:#666;-
6 |     }-
7 |     textarea{-
8 |         height:70px;-
9 |     }-
10 | ul.gfield_checkbox{-
11 |     li{-
12 |         input[type=checkbox]{-
13 |             float:left;-
14 |             margin:0px 4px 0px 0px;-
15 |         }-
16 |         label{-
17 |             float:left;-
18 |             width:220px;-
19 |             margin:-3px 0px 0px;-
20 |             font:11px/17px Georgia,sans-serif;-
21 |             text-align:left;-
22 |         }-
23 |     }-
24 | }-
25 | }-
26 | &.col_2{-
27 |     margin-right:0px;-
28 | }-
29 | }-
30 | }-
31 | .gform_footer{-

```

How not to nest your Sass

The screenshot above is taken from a project I was asked to do some maintenance on a few years ago. I've made the tab characters visible so you can see just how many levels of depth are in use here. This is bad.

To make matters worse, this file is named `_homepage.scss` and contains 1374 lines of Sass. Every single selector is nested within one that starts

```
body.homepage { }
```

Which means that the 1374 lines of code in this file will only ever apply to the homepage of the site in question.



Watch Out!

Don't over-nest your Sass. As a guideline, never go more than 3 levels deep. This will keep your specificity low and increase the chance that styles can be reused.

Sass may be awesome and it may give you superpowers, but with great power comes great responsibility. Here's an exercise to help you be more aware of the issue of nesting.



Exercise: Don't ignore the compiled output

Even though Sass often works its magic in the background and compiles automatically, it's a good practice to look at the compiled styles from time to time, especially when learning. Do check that the CSS that's being output is **CSS you would have written yourself**. If you're seeing long selector chains or lots of repetition, chances are your code could be refactored to DRY it up and reduce bloat. A great tool for understanding the difference between the Sass you write and the CSS that's compiled is [Sassmeister](#).

Take one of the code snippets from this book and paste it into Sassmeister to see the compiled output side by side.

Mixins

We've looked at some of the most popular Sass features already and you could be a perfectly good Sass developer by just combining minification, partials, variables and a dash of nesting. But the Sass compiler can do so much more. Let's wrap up this chapter by looking at just one last feature which will enable you to systemise and componentise your code even further.

Static Mixins

Variables store a single value for later use but sometimes there are times when it's desirable to reuse an entire block of code containing a number of property *and* value declarations. Mixins provide that solution and are used to define a group of CSS declarations that can be included in multiple places in a file or throughout a whole project.

Mixins come in two parts: first we declare the mixin:

```
@mixin name-of-mixin {  
  // mixin styles  
}
```

After a mixin is declared, it can be used

```
.box {  
  @include name-of-mixin;  
  
  // other styles for .box  
}
```

An example of this could be some common text styles that are shared between multiple elements or components. Perhaps our design has a common pattern of uppercase text with increased letter spacing. We could define a mixin to handle this:

```
@mixin uppercase-letter-spacing {  
  text-transform: uppercase;  
  letter-spacing: 2px;  
}
```

This mixin could then be used across various partials:

```
// in _base.scss  
h3, h4, h5, h6 {  
  @include uppercase-letter-spacing;  
}  
// in _homepage.scss  
.intro-title {  
  @include uppercase-letter-spacing;  
}  
// in _contact-us.scss  
.contact-us-button {  
  @include uppercase-letter-spacing;  
}
```

This method of defining and using mixins (which I call *static mixins*) is all well and good but there's little difference compared to just creating a re-usable class to apply these styles on any given element:

```
.uppercase-letter-spacing {  
  text-transform: uppercase;  
  letter-spacing: 2px;  
}
```

```
<h1 class="uppercase-letter-spacing">lorem ipsum</h1>
```

As a general rule, I try to avoid using static mixins as it leads to repetition in the compiled CSS that's unnecessary and could be better achieved with applying a reusable class direct in the HTML.

**Tip: Use gzip compression**

The issue of repetition in the compiled CSS file can be greatly helped by [turning on gzip compression on your server](#) but reducing duplication in the first place is a good practice. Even if configuring servers isn't your responsibility, check that compression is in effect to ensure the best performance of your project.

Dynamic Mixins

Mixins can be made much more useful and much more powerful if they are *dynamic mixins*. These mixins have the ability to change the code that's output by accepting a number of parameters.

**Code Along: Dynamic Mixins**

Follow along with the steps below to create a more flexible text formatting mixin than the static one above. So you can easily see the difference between the Sass code and the compiled CSS, I recommend using [Sassmeister](#).

Defining a dynamic mixin

We can make our text formatting mixin more flexible by allowing the user to define the font-size and decide the amount of letter-spacing to set under different circumstances. If this mixin was to be used with many different font-sizes, the amount of letter-spacing would need to be tweaked to ensure the letters don't appear too spaced out or too tightly packed.

In your Sass, add the static mixin code below as a starting point

```
@mixin uppercase-letter-spacing {  
  text-transform: uppercase;  
  letter-spacing: 2px;  
}  
h1 {  
  @include uppercase-letter-spacing;  
}
```

Add mixin parameters

To allow the user to set the `font-size` and `letter-spacing` at the time of including the mixin, update the code as follows:

```
@mixin uppercase-letter-spacing( $letter-spacing, $font-size ) {  
  font-size: $font-size  
  text-transform: uppercase;  
  letter-spacing: $letter-spacing;  
}
```

The mixin now accepts two parameters, one for the spacing and one for the font size. These parameters can be named whatever you like but always ensure they're clear and would make sense to anyone who might work on the code.



Error: Required parameters

If you're using Sassmeister or compiling your code in your local dev environment, you will notice that this code throws an error until the mixin `@include` statement is updated to use these parameters. We'll look at making the parameters optional very shortly.

Update the `@include` statement

To ensure the compiler runs without errors, update the `@include` statement to pass two comma separated parameters.

```
@include uppercase-letter-spacing( 2px, 2em );
```

Note the compiled CSS that's generated. The first parameter of `2px` is set for the `letter-spacing` and the second parameter of `2em` is set as the `font-size`.

Compiled CSS

```
h1 {  
  font-size: 2em;  
  text-transform: uppercase;  
  letter-spacing: 2px;  
}
```

Set up default parameter values

We can make this mixin more flexible by making the values optional. Perhaps we won't always want to set a font-size or letter-spacing but at the moment the parameters are *required*. We can make them optional by specifying a default value in the mixin declaration as follows:

```
@mixin uppercase-letter-spacing( $letter-spacing:2px, font-size:null ) {  
  font-size: $font-size  
  text-transform: uppercase;  
  letter-spacing: $letter-spacing;  
}
```

By setting these default values, the mixin can now be included in any of the following ways:

```
// use defaults  
@include uppercase-letter-spacing;  
// just set letter spacing  
@include uppercase-letter-spacing( 5px );  
// set both spacing and size  
@include uppercase-letter-spacing( 5px, 2em );
```

**Tip: null variables**

If a variable has the value `null` and is applied to a CSS property, that property won't be output in the compiled CSS. Even though the browser would ignore the value "null" value as a syntax error, this behaviour ensures that the compiled CSS file isn't bloated with unnecessary errors.

Use-cases for Mixins

Mixins are very powerful when combined with parameters as we've just discussed. But this is a very different way of thinking about CSS so here's a list of suggestions where mixins may come in handy.

- Dealing with vendor prefixes
- Outputting breakpoints for responsive design
- Using clearfix in CSS rather than cluttering the HTML with utility classes
- Creating multiple colour schemes
- Adding common blocks of styles for things like vertical centring or horizontal navigations
- Image replacement techniques

Mixins are incredibly useful but again, just like with nesting, be mindful when using them and always keep an eye on the compiled CSS to ensure it's something you would write yourself if you weren't using Sass to do all the hard work for you!

Invisible Code

In the section on Sass folder structure, I mentioned that I have an `includes` folder which is comprised of partials that generate little or no CSS when compiled.

Having now learned about mixins and variables, we can tie off that open ended statement. When we create variables and define mixins, no CSS is generated at compile time; variables and mixins only exist in the Sass world, not in CSS land. When variables are *used* or when mixins are *included* then CSS is generated.

The `includes` folder mentioned earlier is a place to keep partials that contain commonly used variables, mixins, functions, utilities and other reusable modules that may be called upon by one or more compiled stylesheets.

Keeping these useful tools in their own folder makes it easy to call on them when necessary and helps to further modularise a project and keep the process of writing Sass as smooth and as enjoyable as possible.

Wrap Up

Thanks for joining me on your journey to get up and running with Sass. I hope you've enjoyed reading about Sass and working through the exercises as much as I've enjoyed using it and writing about it. If you've just been skimming, here's the `tl;dr`

- Sass is a CSS preprocessor.
- We write Sass and it's *compiled* into CSS.
- Sass was invented by Hampton Catlin back in 2006 and is currently maintained by Nathalie Weizenbaum.
- Sass enables us to author styles in a developer-friendly way but package them for optimised delivery to the browser.
- We can split up our code into multiple files to improve readability and maintainability. Sass combines everything together at compile time.
- Variables allow us to label values for greater meaning and allows values to be reused multiple times across a project.
- Nesting selectors means we can write less and provide more hierarchy to our styles. However, we don't want to nest more than three levels deep.
- Mixins allow us to reuse larger blocks of code and mixins that accept parameters enable variations to those blocks of code to be quickly generated.
- All in all, Sass is awesome.

Next Steps

So now that you're up and running and Sass is quite literally watching for changes and awaiting your next move, what's next?

Practice what you've learned.

Ensure you've tackled the practical exercises in this book as the best way to learn is by doing. Take a past project - perhaps a side project or some work you did for a client - and covert the whole thing over to Sass and use your knowledge of partials, variables, nesting and mixins on something you know well.

If you're feeling more adventurous, make a commitment to use Sass on your next project. You may work slower to begin with but the learning curve will be over quickly. In just a matter of days (perhaps as little as a few hours) you will have probably far surpassed your typical productivity with "vanilla" CSS. Embrace the challenge and you'll go far; sharpen your skills and get more value from your time.

During your work with Sass, do refer back to this guide but also check out the [Sass Website](#) and have a read of the documentation to see what else is possible or to jog your memory of the various features and syntax.

Never Stop Learning

Practice makes perfect but there is still lots to learn. I've been using Sass almost daily for many years and I still pick up new tips and tricks on a weekly basis (I even picked up the tip about `null` variable values whilst writing this book!).

To share my knowledge I'm producing a totally free, 26-part video series all about Sass for AtoZ CSS. Each video will take a single Sass concept - one for each letter of the alphabet - and dive deep into it in a compact, highly produced 5-10 minute educational screencast. The series launches in March 2016 and to get access and receive updates on new videos and weekly CSS and Sass news, you can sign up for the mailing list at www.atozsass.com.

I look forward to sharing more Sassy goodness with you very soon and I wish you all the best with your future projects!

Cheers,

– Guy

Thank You For Reading

Thanks for reading *Up and Running with Sass*, a practical introduction to CSS preprocessing.

I want to ensure you get the most out of the time you spend learning and I'm always looking to improve my work and help you succeed in your career.

Do let me know what you thought of this guide. If you found a typo, I'll fix it. If something didn't make sense, I'll try to explain it more clearly. As a purchaser of this book, you'll receive any future updates for free so you and other readers will benefit from your comments.

What Are You Struggling With?

What else can I help you with? What's the hardest thing about your job? What would it take to get you to the next level? What do you want to do but just don't know where to start? Send me an email to mail@guyroutledge.co.uk and let me know. **I read every message personally** and would love to help you succeed.

If You Found This Book Valuable

I'd love your help to spread the word so you can help others discover it and improve their skills too. There are a number of ways you can help others find this book:

1. Follow me on Twitter [@guyroutledge](#) or [@atozcass](#).
2. Send a tweet. "[I just finished Up and Running with Sass by @guyroutledge and it's awesome!](#)".
3. Tell your teammates about the book.
4. Visit [AtoZ Sass](#) and sign up for the mailing list. You'll get weekly CSS & Sass news and be the first to hear about new blog posts, videos, books and courses.
5. Send me a testimonial to mail@guyroutledge.co.uk about how this book helped you.

Thanks again. You're awesome.

Cheers

– Guy

About the Author

I made my first “website” back in the good old days of font tags and scrolling marquees. I use the term “website” quite loosely because by modern standards, it certainly didn’t look like one.

CSS was initially released in December 1996 and while I first dabbled with web design in late 1997, the concept of CSS certainly hadn’t reached the IT department at my school, where I was first exposed to the internet and wrote my first line of HTML.

I enjoyed playing around and making things with code but it was really just a bit of fun. I continued to dabble in HTML and CSS throughout University and made a number of iterations of the website for my first business; I worked as a freelance cameraman in the TV and film industry for five years. During that time, I made a couple of simple portfolio sites for actor friends of mine.

Film making wasn’t the most financially viable career so I transitioned sideways and went to work with a friend of mine who exported prestige cars to the Far East. I did that for a couple of years whilst searching for my next career move.

It took me a surprising amount of time to remember that I really enjoy making things with code and overnight I decided to quit my job and immerse myself in learning so I could train myself up to industry standard. I read lots of books, watched a lot of screencasts and worked through a whole array of online tutorials on design, HTML, CSS and jQuery - I’ll confess that I never learned “proper” JavaScript first but have remedied that now.

Over the last few years I’ve worked on a vast array of projects for a wide variety of clients, agencies, startups and small businesses. I love what I do any love sharing my knowledge and experiences through teaching. I’ve produced videos and written for top industry sites like Tuts+, Sitepoint and Code School and on my own platform, AtoZ CSS. I also teach courses and workshops in person in London and still take on client work through my own consultancy business.

Moving in to web development was the best decision I’ve ever made (I wish I’d done it sooner!) and I feel so fortunate to have landed on my feet. But more than that, finding something that I’m good at, that other people also want to learn about has given me the opportunity to teach. Sharing knowledge, tips & tricks and resources in the classroom became something I enjoyed so much, that I decided to try my hand at online teaching and that’s where AtoZ CSS came from. It’s been a lot of hard work but it’s already starting to pay off and I have the opportunity to help people like you succeed and thrive.



Guy Routledge