Tutorial to Compass and SASS Documentation

Release 0.3

Milan Darjanin

May 31, 2013

CONTENTS

CHAPTER

ONE

FIRST STEPS TO THE SASS

1.1 History

Sass (Syntaticly awesome stylesheet) is meta-language created on top of CSS. It's main purpose is to provide more enhanced features to the CSS that are useful for creating manageable stylesheets. It was originally created by Hampton Catling (http://www.hamptoncatlin.com/). He and Nathan Weizenbaum (http://nex-3.com/) designed Sass through 2.0 version. Nathan is the primary designer of Sass and the main developer since its inception. In late 2008 joined the Sass team Chris Eppstein (http://chriseppstein.github.com/). Chris and Nathan designed Sass from version 2.2. Another accomplishment of Chris is the Compass (http://compass-style.org/), the first Sass-based framework.

The official implementation is done in Ruby. And through this manual I will be using only this one. There are attempts to make Sass interpreter in Javascript so you can run it on the server with Node.js or PHP version named PHPSass (http://www.phpsass.com/). Sass is available under the MIT License (http://sass-lang.com/docs/yardoc/file.MIT-LICENSE.html).

1.2 Installation

Now after few words from history is time to move on. Before we can start with the syntax, it's necessary to install the Sass interpreter. How I said before we will be using Ruby implementation. And because we have many operating systems with different dependencies I will

give you hints how to prepare your system.

Windows

The Windows does not come with Ruby installed at default. The fastest way how to install it is to download RubyInstaller (http://rubyinstaller.org/downloads/). When it is done, go to Start Menu, Accessories and run Command Prompt. But faster would be to press *Win+R* and put in the *cmd* command and press Run. Next step is same for all platforms.

Linux

This category is more general, while there are many distribution with Linux kernel. But the most used systems today have roots in Debian. In this group you can add Debian, Ubuntu, Linux Mint and so on. If you are using Arch, then I don't think think you need to read how to install Ruby.

For Debian based systems you just need to start Terminal and run in it

```
sudo apt-get install ruby1.9.1
```

It will ask your password and after that it will install Ruby.

MacOS X

Your new Mac comes with Ruby installed. So only thing that you must do is to find your terminal. You can use Spotlight and write in it Terminal. And run it.

Last step

At the end to install the Sass gem just write following command into the terminal or command prompt. When the installation ends, you are done.

```
gem install sass
```

1.3 Hello World example

It's good idea to try if it works. Run in terminal or command prompt

```
sass --scss
```

Your input would be

```
$header-color: #fe3242;

h1 {
    color: $header-color;
}
```

And when you are done press *Ctrl+D*. You should get

```
h1 {
    color: #fe3242;
}
```

How can you see, the line starting with dollar sign disappeared and the color value has changed to value defined for *$header-color*. This is simple example of using variables in Sass. When there was no problems you can move to the next chapter.

1.4 Compiling Sass

In "Hello World example" the code runs in the interactive mode of the Sass, but in real life you write code into files. For that sass console application has options that helps with that. There exist GUI applications for working with Sass, but I'm a CLI guy and I think you will like it too.

The easiest way how to convert *.sass* or *.scss* file to the *.css* version is using following command.

For files

```
sass --watch input.scss:output.css
```

For whole directories

```
sass --watch input-dir:output-dir
```

CHAPTER

TWO

SASS SYNTAX

After details how to setup up your working environment for Sass, it's time to move on to the syntax of this language. How I said earlier anything written in CSS is valid Sass code. It's not always true. The thing is that Sass has two possible syntaxes. The older one, called simply Sass with extension *.sass*, takes inspiration in Haml (http://haml.info/). There are no semicolons, no curly brackets and few more differences from style that will be used. The important thing in it is using indention. If you met with languages like Ruby or Python, than you understand. For people who have no clue what I'm talking about, is here small example.

First is the code in Sass (.sass extension)

```
#main
    background: red;
    color: white
    a
        font:
            weight: bold
            size: 2em
            family: serif
        color: yellow
        &:hover
            color: green
```

And now CSS equivalent to code above.

```
#main {
    background: red;
    color: white;
}
#main a {
    font-weight: bold;
    font-size: 2em;
    font-family: serif;
    color: yellow;
}
#main a:hover {
    color: green;
}
```

This approach to the syntax has some advantages and if you have some experience with languages where indention is so important than go for it. But in this materials I will be using most often the SCSS (Sassy CSS) syntax. It's more similar to CSS so there would be no problems to start using Sass, what is main purpose of this tutorial.

CHAPTER

THREE

VARIABLES

How often happened to you that you were writing CSS, in which you need to set up color for some element, but you don't remember the code of used color? You can still find it in document, but it could take some time. Or another example. You got some code at which had worked other developer and only thing that you need to do is to change colors of all links in the document. The problem is that you don't know in how many declarations is that color used in document and how we can see later, code written in Sass is often divided in many files. If the previous developer used variable to store the color value, than your work would be just to find the declaration of color for link and change it.

In this example situations variables come to be handy. It's true that they are often used as constants in Sass. There are no problems to change their values later, but it is not good practice to do so, while it can start to be mess and you can not be sure which value is used at the moment so easily. The definition of variable starts with symbol *$* following with the variable name, double-colon and the variable value. The value can be color code in any format supported in CSS, string, number or length with unit.

```
$color-var-name: rgba(42,42,42,1);
$length-var-name: 960px;
$string-var-name: "|";
$number: 0.2;

#main {
    width: $length-var-name
}
```

```
a {
    color: $color-var-name;
    opacity: $number;
}

#main {
    width: 960px;
}
a {
    color: rgba(42,42,42,1);
    opacity: 0.2;
}
```

Note: Naming conventions They are inherited from CSS. The name for variable should be created from alphanumeric symbols and separated by hyphen. The name should say enough about the value that is saved in it. Try to avoid names like *$red-color* and than use it for all your links. Better approach is to create color scheme like *$red: #E03838;*,than create *$link-color: $red;* and use it for links. If you came to state that you need to change the color from red to green, you will just declare *$green* and set the *$link-color* to it. It's better, because if you stay with *$red-color*, than you will probably change the value stored in *$red-color* to green and it does not make sense.

CHAPTER

FOUR

NESTING

I'd like to start with simple CSS code for horizontal navigation.

```
nav { position: absolute; right: 5em; bottom: 2em; }
nav ul { list-style: none; }
nav ul li { display: inline; }
nav ul li a { color: #4590DE; text-decoration: none; }
nav ul li a:hover { text-decoration: underline; }
```

You probably met with similar code. If you look at it you can see that I repeated some selectors. In final style they are important, but when you are writing code, you don't want to repeat yourself. Computers are good for repetitive work so why don't use them for this too? Sass has solution for this. It's called nesting. It's main idea is that child elements are written inside of the parent curly brackets. Than I can rewrite the CSS code into.

```
nav {
    position: absolute;
    right: 5em;
    bottom: 2em;
    ul {
        list-style: none;
        li {
            display: inline;
            a {
                color: #4590DE;
                text-decoration: none;
```

```
                &:hover {
                    text-decoration: underline;
                }
            }
        }
    }
}
```

The indention in code is not important, but it's recommended, for easier reading of the code. How you can see I didn't repeat any selector. There is interesting syntax with the ampersand. *&:hover*. The ampersand stands for the parent selector. The reason why I didn't used it for previous declaration is that it's added there automatically. So if you write

```
nav {
    ul {

    }
}
```

it can be rewritten using *&*-syntax to

```
nav {
    & ul {

    }
}
```

We need to refer on the parent selector in case that there is no need for space between selectors. For example when we use pseudo-classes or pseudo-selectors.

```
table {
    &.users-mode {
    }
}
```

will be translated to

```
table { }
table.users-mode {  }
```

There is one more way where to use nesting. And it's for CSS properties. Some of them are created with some prefix like font-, text-, border-,etc. and if you are going to set more of them you can use the short version, but sometimes you need to specify it more explicit. And than comes nesting handy. The example would be best for it.

```
.block {
    border: {
        width: thin thin 0 0;
        color: red blue;
        style: solid;
    }
}

.block {
    border-width: thin thin 0 0;
    border-color: red blue;
    border-style: solid;
}
```

CHAPTER

FIVE

SASSSCRIPT

Extra features that you don't find in CSS brings the SassScript. It allows to use arithmetic operations, interpolation and functions. If you want to just try some of it without writing files, than for you is there Interactive Shell.

```
sass -i
>> 1px + 1px + 1px
3px
>> #123 - #010101
#122334
>> #777 + #888
white
```

5.1 Data types

The SassScript supports 6 data types. There is no need to declare them. It will be automatically done. They are

- numbers, e.g. 1.2, 13, 10px
- strings with and without quotes, e.g. "cube", 'triangle', line
- colors, e.g. red, #123456, rgba(234,123,0, 0.8)
- booleans, e.g. true, false
- null

- list of values separated by spaces or commas, e.g. thin solid black

You don't need thing about these types a lot. Only in cases that you store for example string into variable, but you want to use it to set a size of font. In this case it doesn't make sense.

5.2 Functions

The classic CSS does not have many functions. One of example could be the *calc()* function for the arithmetic operations, but it's supported not in all browsers.

SassScripts brings more handy functions. For the full reference and examples to them I recommend the official documentation SassScript Functions Reference (http://sass-lang.com/docs/yardoc/Sass/Script/Functions.html). You can find there all functions with short example. For that reason I'm not going to copy it. I give you in moment one example where can be functions useful.

You can sort them base on the target area of use.

- **Color functions**
  - RGB
  - HSL
  - Opacity
  - Other
- String functions
- Number functions
- List functions
- Introspection functions

Generating a color palette

Nice example of using the functions is generating color palette. You need only to set the base color. In case that you would change it for any reason, other colors will change depending on it. Experimenting with the color palette is simple than.

```
$base: #633;
$complement1: adjust-hue($base, 180);
$complement2: darken(adjust-hue($base, 180), 5%);
$lighten1: lighten($base, 15%);
$lighten2: lighten($base, 30%);
```

CHAPTER

SIX

@-RULES AND DIRECTIVES

Sass supports all CSS @-rules like *@import*, *@media* or *@font-face*, but some of them extend and gives them more power.

6.1 @import

With import rule you will meet often using Sass. It extends CSS import rule, so you can import *.scss* and *.sass* files. The output will be merged into one single CSS file and all variables and mixins defined in the imported files will be available in the main file. With this behavior you can split your styles into smaller files defining specific elements. It makes easy to append or edit the code.

There are some special circumstances at which will the @import rule works like the CSS.

- The file's extension is *.css*.
- The filename begins with *http://*.
- If the filename is *url()*.
- If the *@import* has any media queries.

```
@import "cube.css";
@import "cube" screen;
@import "http://cube.edu/style";
@import url(cube);
```

```
@import "cube.css";
@import "cube" screen;
@import "http://cube.edu/style";
@import url(cube);
```

If we want to import the file *cube.scss* we can write

```
@import "cube.scss";
```

or just simply

```
@import "cube";
```

If you want to import more files, it's possible to write

```
@import "first", "second";
```

If you name the *.scss* file with underscore before filename *_cube.scss*, than it's code will be added to the main *.css* file, but it will be not compiled to CSS at own. In *@import* you don't need to write the underscore, but it's important that in the same folder can not be more files with the same name. (If in folder is *cube.scss*, than you can not use *_cube.scss*). This type of naming of files is called partials.

One of earlier mentioned features of Sass is nesting and it's possible to use it with *@import*. Most of time will you use the *@import* at the top of the document. But there can come situation when it would be handy to include whole another file into some class. At that case you can call *@import* under class.

The best way to understand is through example. The *box.scss* and *screen.scss* contain following code

```
// Content of the box.scss
.box {
    color: red;
    .button {
        background: #444;
    }
}
```

```
// Content of the screen.scss
.screen {
    @import 'box.scss';
}
```

The compiled version is

```
.screen .box {
    color: red;
}
.screen .box .button {
    background: #444;
}
```

There are few exceptions. There exists directives that can be only at the base level of the document. So if you are calling *@import* into selector than the imported file can not contain *@mixin* and *@charset*. It's not possible to @import in mixins and control directives.

6.2 @media

@media directive can be used as defined in the plain CSS, but it has one extra capability - it can be nested in CSS rule. If it appears nested, than it bubble to the base level, containing all selectors in which it's included. This approach helps to make your code readable if you are using the @media.

```
.sidebar {
    width: 300px;
    @media screen and (orientation: landscape) {
        width: 500px;
    }
}

.sidebar {
    width: 300px;
```

```
}

@media screen and (orientation: landscape) {
    .sidebar {
        width: 500px;
    }
}
```

This way you don't break the flow of your selectors that are nested into each and again you don't need to repeat to writing the selector that you want to specify with @media. You can complain that you often write the value for the *@media*, but we have variables. This is nice place where you can use it.

```
$landscape: 'screen and (orientation: landscape)';

.sidebar {
    width: 300px;
    @media #{landscape} {
        width: 500px;
    }
}


.content {
    width: 400px;
    @media #{landscape} {
        width: 600px;
    }
}

@media screen and (orientation: landscape) {
    .sidebar {
        width: 500px;
    }
    .content {
        width: 600px;
```

```
    }
}

.sidebar {
    width: 300px;
}
.content {
    width: 400px;
}
```

You can define more @media properties for specific devices at the start and if you need to change some properties you don't need to look through documents where you write device-specific rules and no selectors were written twice.

6.3 @extend

There are often cases when you need to use the all rules from one selector and add only some new. Most used way how to do that is using some general class and than more specific class that sets the different properties. Than the HTML will be following

```
<div class="error error-login">
    Sorry, bad login or password. Try it again.
</div>
```

The css to the code will be

```
.error {
    border: thin solid #FF5151;
    background-color: #F9E9E9;
}

.error-login {
    border-width: thick;
}
```

This method is functional and it's often used, but you must not forgot the error class. The *@extend* directive helps to avoid the some problems that are possible using this way. Than in the HTML will be written

```
<div class="error-login">
    Sorry, bad login or password. Try it again.
</div>
```

```
.error {
    border: thin solid #FF5151;
    background-color: #F9E9E9;
}
.error-login {
    @extend .error;
    border-width: thick;
}

.error, .error-login {
    border: thin solid #FF5151;
    background-color: #F9E9E9;
}
.error-login {
    border-width: thick;
}
```

@extend works by inserting extending selector anywhere the extended selector appears. For better illustration I add example.

```
.error {
    border: thin solid red;
    padding: .5em;
    color: red;
}
.error.icon {
    background: url('images/error.png');
}
```

```
.error-login {
    @expand .error;
    font-weight: bold;
}

.error, .error-login {
    border: thin solid red;
    padding: .5em;
    color: red;
}

.error.icon, .error-login.icon {
    background: url('images/error.png');
}

.error-login {
    font-weight: bold;
}
```

If you are familiar with Object Oriented languages as Java or C++ you sure know the meaning of abstract class or function. There are not allowed instances from abstract classes, so they must be inherited by another class. And in Sass exists similar way how to define the selector. It's called "placeholder selectors". They are defined in Sass version of code, but they are not compiled to the CSS. Only if they are extended by another selector. It helps to avoid names collisions and the in the output CSS they show up only if they are needed. They are most of time used if you are creating framework. The syntax difference from the selectors for classes and ids only in first characters. You don't use the . or #, but %. So "placeholder selector" for the error could be *%error*. Everything else works like it is normal selector.

The main difference between using extend and mixins is in the output CSS. Say that we want to create four buttons and only the color of the background would be changed. If you use for that mixins the output would have the same code for every button generated, and only the color codes would be different. On other hand, if you extend generic "placeholder selector" for buttons and set for each one only different color, than the generated CSS will have shorter code. In situation when you want to load your site as fast as possible is this approach good

idea. But always this things depends on the situation.

If you want to use *@extend* inside of the media block, there is some restrictions. You can extend only selectors that are inside of the media block.

CHAPTER

SEVEN

MIXINS

Some of many advantages of Sass is keeping your code readable and don't repeating yourself. For the DRY (don't repeat yourself) exists mixins. If you work with programming languages before, you can say that they look like functions. But like many things in Sass comes from "Ruby universe", mixins exists there too. The way how mixins work is to include their code at the place where they were called.

Mixins are defined with the directive *@mixin* following with the name of mixin and optionally the arguments. After that is there a block containing content of mixin closed into curly brackets.

```
@mixin button {
    border: thin solid #40AECA;
    background: #85C7D8;
    border-radius: 5px;
    color: white;
    &:hover {
        background: #7EB7C6;
    }
}
```

If you try to compile this with sass, than the output will be empty. The reason is that you don't use the mixin. And the second thing what you can see at definition of mixin is *&:hover*. We met with it in nesting, but there we knew who was parent. Here will be the parent selector defined at the moment of calling the mixin.

For inserting the content of the mixin use *@include* directive.

```
.button {
    @include button;
    height: 30px;
}

.button {
    border: thin solid #40AECA;
    background: #85C7D8;
    border-radius: 5px;
    color: white;
    height: 30px;
}
.button:hover {
        background: #7EB7C6;
    }
```

But this is not all what comes with mixins. I give you example when you need to have same styled buttons, but with different background colors. You can define the new color after *@include button*, but there comes some repeating work. You must always define the new behavior for the hover state too. All because of using different color. How can we improve it? We can use the arguments that would be passed to the mixin. The best way how to explain it would be with example.

Say that you want to create three different color buttons. One would be normal with light gray background and would be for classic actions. Next on would be the error button that has red background and finally information button with blue background. And we want to define one mixin and then change just colors when we include it.

```
@mixin button($color) {
    border: thin solid $color - #222222;
    background: $color;
    border-radius: 5px;
    color: white;
    padding: 5px;
```

```
    &:hover {
        background: $color - #161616;
    }
}

.button {
    @include button(#B1B1B1);
}

.error-button {
    @include button(#FB4242);
}

.info-button {
    @include button(#549EE5);
}

.button {
  border: thin solid #8f8f8f;
  background: #b1b1b1;
  border-radius: 5px;
  color: white;
  padding: 5px; }
  .button:hover {
    background: #9b9b9b; }

.error-button {
  border: thin solid #d92020;
  background: #fb4242;
  border-radius: 5px;
  color: white;
  padding: 5px; }
  .error-button:hover {
    background: #e52c2c; }
```

```
.info-button {
  border: thin solid #327cc3;
  background: #549ee5;
  border-radius: 5px;
  color: white;
  padding: 5px; }
  .info-button:hover {
    background: #3e88cf; }
```

Knowledge of this techniques is enough for you to start using the Sass on daily basis. There exists some more advanced things that comes handy, but their main purpose is for make more flexible code that can be part of framework like Compass. If you work on large projects and you use some styling techniques often, than it could be good idea to invest some time to write them into simple framework for you and use it in your projects, but before you start writing everything on your own, it could be good idea to jump to the chapter about Compass and look if things that you need do exist in it.

CHAPTER

EIGHT

CONTROL DIRECTIVES

SassScript supports control directives for including styles only under specific condition or including same style several times with variations. Their main purpose is to use them in mixins, those that are part libraries like Compass and requires flexibility.

8.1 @if

IF is one of the basics directives for control the flow. The style would be applied only if the condition returns anything else than *false* or *null*. In conditions are allowed logical operations *and* and *or* that require at least two conditions and the negation *not*.

```
p {
    @if 1 + 1 == 2 { border: 1px solid; }
    @if not (5 > 3)    { border: 2px dotted; }
    @if null             { border: 4px dashed;}
}

p { border: 1px solid; }
```

At case that you need to check if the variable content is one of many, than comes handy the *@else if*. The last must be *@else*.

```
$language: ruby;
p {
    @if $language == python {
```

```
        background: green;
    } @else if language == c# {
        background: blue;
    } @else if language == ruby {
        background: red;
    } @else {
        background: yellow;
    }
}

p { background: red; }
```

8.2 @for

In case that you need to repeat some action with different value in the output, you can use *@for* loop. It sets the value in variable from starting point to end. There are two forms of for-loop in Sass. First is *@for $var from <start> through <end>* and the second is *@for $var from <start> to <end>*. The variable *$var* is normal variable that can be named how you need. It's common to name it *$i*. The *<start>* and *<end>* can be any expressions that returns integer. The difference between these two forms is in the *trough* and *to*. If you use *through* the *<end>* value will be used at the end. If you use *to* the loop stops at the *<end>-1* value.

```
@for $i from 1 to 4 {
    .item-#{$i} { width: 2em * $i; }
}

.item-1 { width: 2em; }
.item-2 { width: 4em; }
.item-3 { width: 6em; }
```

8.3 @each

The for-loop is good if you are working with numbers. But if you want to work with list of words, than using the *@each* is better decision. The syntax for each is simple. *@each $var in <list>*. The variable *$var* is working the same way how in the for-loop. So in every step the *$var* value is equal to one of the items in the list.

```
@each shape in circle, triangle, square {
    .#{shape}-icon {
        background-image: url('/images/#{shape}.png');
    }
}

.circle-icon { background-image: url("/images/circle.png"); }
.triangle-icon { background-image: url("/images/triangle.png"); }
.square-icon { background-image: url("/images/square.png"); }
```

8.4 @while

If the step one that is in the for-loop is not good for you, than you must use the while. It will run until the condition is true. The syntax is

```
@while condition {
    // code
}
```

And example

```
$i: 6;
@while $i > 0 {
    .box-#{$i} {
        width: 2em * $i;
        $i: $i - 2;
    }
}
```

```
.box-6 { width: 12em; }
.box-4 { width: 8em; }
.box-2 { width: 4em; }
```

CHAPTER

NINE

COMPASS

At this point you should know enough about Sass and what is possible to do in it. There are many ways how to use it. You can use the approach of writing everything for yourself and only if you need it. There are probably some of you who always work this way. But many developers want to save time to start with next project. They often take some framework, that has the common things written and they just write the new parts specific for their project. The community about Sass is not different. There is not just one framework that you can use, but in this tutorial I will talk about the oldest and probably the most used of them. Compass.

If you have some time to spare and learn something more, than I recommend to go through the code of the Compass, that you can find in it's public repository on the GitHub: github.com/chriseppstein/compass (https://github.com/chriseppstein/compass). There are more than just Sass code, while Compass has own tools to create projects. But important for you is to target the framework and then compass folder, where you can find only styles written in Sassy CSS (.scss).

9.1 Installation

Installation of this framework is simple. Important is to have installed Sass. If you don't, for some reason, than go to the chapter about Sass, where is written the guide for installation. If you are back or had the Sass installed, than put into console following command:

```
gem install compass
```

There is possibility that on the UNIX/Linux systems will you need to add *sudo* before the command, while it will want to write into protected folders own data. It will download and set up path to compass files and prepare it to using for you.

CHAPTER

TEN

WORKING WITH PROJECTS

In every project in which you want to use the Compass, you need to have configuration file, that says to the compiler where to look for files, where to save the output and some other options important for work. For all that there is command line tool called simply *compass*. I'm not going through all it's options, only the primary. If you want to see all options use option for help. In case that you need something, that it's not written in following text, I recommend the Compass Tutorials (http://compass-style.org/help/tutorials/), where are covered even more advance topics.

```
compass -h
```

Now if you have some time try to look at the list of all options that compass has. I'm going to explain the primary ones and what they do.

10.1 create

The most important is to create new Compass project. It's done with the option *create*. After the option follows the name of the project.

```
compass create HelloWorld
```

The start of its output will be following

```
directory HelloWorld/
directory HelloWorld/sass/
```

```
directory HelloWorld/stylesheets/
create HelloWorld/config.rb
create HelloWorld/sass/screen.scss
create HelloWorld/sass/print.scss
create HelloWorld/sass/ie.scss
create HelloWorld/stylesheets/print.css
create HelloWorld/stylesheets/screen.css
create HelloWorld/stylesheets/ie.css
```

It creates the folder HelloWorld with *config.rb*, in that is stored the settings that will be used by the compass compiler, and two folders one for yours Sass code - *sass* and one for generated css called *stylesheets*. It even creates three *.scss* files for your styles and compile them to their CSS versions. At the end of the output is how to include the styles to your HTML or some similar document, where you need the styles, but this you should know.

The names of the files don't need to be equal to the version generated by compass. Only the *config.rb* file name needs to stay without change. It stores the settings for the compiler where to look for the sass files and where save the output. It even sets where are saved images that you use and your javascript if you have any.

Content of the *config.rb*:

```
# Require any additional compass plugins here.

# Set this to the root of your project when deployed:
http_path = "/"
css_dir = "stylesheets"
sass_dir = "sass"
images_dir = "images"
javascripts_dir = "javascripts"

# You can select your preferred output style here (can be overridden via the
# output_style = :expanded or :nested or :compact or :compressed

# To enable relative paths to assets via compass helper functions. Uncomment
# relative_assets = true
```

```
# To disable debugging comments that display the original location of your s
# line_comments = false


# If you prefer the indented syntax, you might want to regenerate this
# project again passing --syntax sass, or you can uncomment this:
# preferred_syntax = :sass
# and then run:
# sass-convert -R --from scss --to sass sass scss && rm -rf sass && mv scss
```

There are comments that explain a lot of things, if you are not sure what does some option mean. You are probably not going to change a lot of the code in here. Only at the time of deployment of the project is nice to uncomment the *line_comments* (just delete the hash sign #) and set the *output_style* to *:compressed*. This way will compass generate for you the smallest version of your style.

10.2 init

If you had started some project and later you figure out that you need to use Compass, than comes the *init* option handy. It create all important files for compass in the directory where is your project. Enter the folder with your project. For example, it's called *web_app* and run following command.

```
compass init
```

After that you are done. It created *config.rb* and the folders for the sass styles and generated CSS. You can say that you can create it on your own, while it's just one configuration file and few folders. But *init* makes it for you.

10.3 compile

Probably the most important option. It looks at the configuration file and base on the settings it generates CSS from your Sass.

```
compass compile [/path/to/project]
```

If you are in the same directory as is the *config.rb* file, than the path won't be there. The path to project is important if you in other directory as the configuration file.

10.4 watch

This option will be used most of the time. It does the same thing as the *compile* with one feature at the top. It waits for changes of your sass code and if there are any, *watch* will compile your code automatically.

```
compass watch [/path/to/project]
```

Again the path is important only in case that you are not in the same directory as is the *config.rb* file.

CHAPTER

ELEVEN

COMPASS CORE

The Compass library is created from two frameworks at the time of writing. The one is the core of the Compass about which would be this section. Another is the Blueprint framework. I'm not going to talk about the Blueprint while it will be deprecated from newer versions of Sass. If you need some toolkit to work with grid and layouts, than I recommend to look at the Susy (http://susy.oddbird.net/). It's build with the Sass and works nice with Compass.

The Compass core is spliced into six parts.

- CSS3 - how the name suggests it contains mixins to make life with CSS3 easier
- Typography
- Utilities
- Layout
- Reset - global reset based on the Eric Meyer's reset 2.0 (http://meyerweb.com/eric/tools/css/reset/index.html)
- Helpers - functions that add new functions to existing in the SassScript

To each one will be dedicated own section where I go through the features that it has. If you need only to find something specific and you don't need comments for it than I recommend the Compass reference at Compass Reference (http://compass-style.org/reference/compass/).

Easiest way how to start using Compass is to use

```
@import "compass";
```

After that you can access everything included in the CSS3, typography and utilities category.

11.1 Reset

All browsers have default styles that are called if the site has not own stylesheet for required element. With one small problem. They have slightly different predefined styles for some elements, so for example the padding for *div* haven't have to be the same. And there are many different places when you need no padding and margins defined. For that most of the developers specify at the begin of the CSS the 'reset' rules. Because this is often done, Compass has built-in support for it. Just at the begin of Sass code include the reset and it will append the global reset based on the Eric Meyer's reset 2.0 (http://meyerweb.com/eric/tools/css/reset/index.html).

```
@import "compass/reset";
```

If you need specific version only for some elements only include utilities from the reset. Most of the time you will use the global reset, if not than at Reset Utilities (http://compass-style.org/reference/compass/reset/utilities/) you can find the reference for mixins for more specified reset.

```
@import "compass/reset/utilities";
```

11.2 CSS3

The CSS3 brings many new features to modern browser. From basic things like border radius through box-shadow to advance effects created with transitions and key-frames. Everything would be great if CSS3 would be completed and all features in it implemented in all browsers the same way. But this is just wish of all web-developers. Truth is that browsers came with their own prefixes for function of CSS3 which has not yet been accepted to the final version or they have their own ideas that they want to make their browser more advance for example.

Every web-developer today must have seen vendor prefixes like -webkit-, -ms-, -mz- and -o-. If you need to create button, that drops shadow, in perfect universe you will write

```
.perfect-button {
    ...
    box-shadow: 2px 2px 5px 4px rgba(42,42,42,0.8);
    ...
}
```

But in this world you must write something like

```
.just-button {
    ...
    -webkit-box-shadow: 2px 2px 5px 4px rgba(42,42,42,0.8);
       -moz-box-shadow: 2px 2px 5px 4px rgba(42,42,42,0.8);
            box-shadow: 2px 2px 5px 4px rgba(42,42,42,0.8);
    ...
}
```

This is one of the better cases when there not all browsers have their own prefixes. But we need to add two more lines just to secure that it will work in as many browsers as possible. Finally the same example written in Sass using Compass.

```
@import "compass/css3"

.scss-button {
    ...
    @include box-shadow(rgba(42,42,42,0.8) 2px 2px 5px 4px);
    ...
}
```

The import is needed only once so I will not count it. We are again at one line for the box-shadow. The code with vendor prefixes will be generated by mixin defined in Compass. I'm not going to write specifically about mixins defined in the Compass, while they can change through time. For the most updated version visit their site with Compass CSS3 reference (http://compass-style.org/reference/compass/css3/).

11.3 Helpers

The Sass comes with huge library of functions and the Compass helpers section extends it. The full reference of it is at Compass Helpers (http://compass-style.org/reference/compass/helpers/). I don't think that you would ever need all of them in one project. But few of them can save a lot work so you will be using them more frequently.

It can happen that through development of the web application are two groups. One is coding the back-end with all logic and interactions and the second is aimed on the front-end and working on the styles. It can happen that at the deployment are made changes to the paths for the images. If you are using just CSS, you must go through code and change the occurrence of the problems. In helpers exists function

```
image-url(path, [only-path])
```

It looks into the *config.rb* file and takes the images_dir value. For example I have image called *smiley.png*, that is saved in the folder *images* in the project.

```
.avatar {
        background: image-url('smiley.png');

        &:after {
                content: image-url('smiley.png', true);
        }
}

.avatar {
        background: image-url('/images/smiley.png');
}
.avatar:after {
        content: "/images/smiley.png";
}
```

How can you see there is slightly difference, if you set the second argument to *true*. Than it will omit the *url()* in the output. The similar functions exist for the fonts. Only difference is in the name *font-url*.

Next useful function is for dimensions of the images. If you need to know the dimensions of the image, compass comes with functionality for that.

```
image-height(image);


image-width(image);
```

The image is relative path of the image. Another tool for images is the function

```
inline-images(image, [mime-type]);
```

I recommend to use it only for small images, like icons, where it can save the HTTP request, with little larger CSS file.

11.4 Utilities

The Compass Utilities brings mixins for some common tasks. There are utility for control what will be in the print version displayed or if you need to fast way to style table that you can use *@import "compass/utilities/tables";* example can be find at Table utility example (http://compass-style.org/examples/compass/tables/all/).

At last I would like to show you how to work with sprites in the Compass. The reason for using sprites is to save HTTP request and speedup the load of the site. They are most of time used if you have many small icons. Without using sprites every one image must be loaded separately. With sprites it loads all images and that using CSS use only needed.

In our example we have all 3 icons for social networks with dimensions 32x32 pixels saved in the directory *social*.

```
images/icons/social/fb.png
images/icons/social/gplus.png
images/icons/social/tweet.png
```

I will show you the basic and easiest way how to create sprite. At the top of your Sass file put

```
@import "icons/social/*.png";
@import all-social-sprites;
```

It will generate

```
.social-fb,
.social-gplus,
.social-tweet { background: url('/images/icons/social-s34fe0232ab.png') no-r

.social-fb { background-position: 0 0; }
.social-gplus { background-position: 0 -32px; }
.social-tweet { background-position: 0 -64px; }
```

The numbers in the name of the generated sprite will you get different. If you want to learn more about the generating sprites in the Compass, than read Tutorial about sprites at Compass (http://compass-style.org/help/tutorials/spriting/).