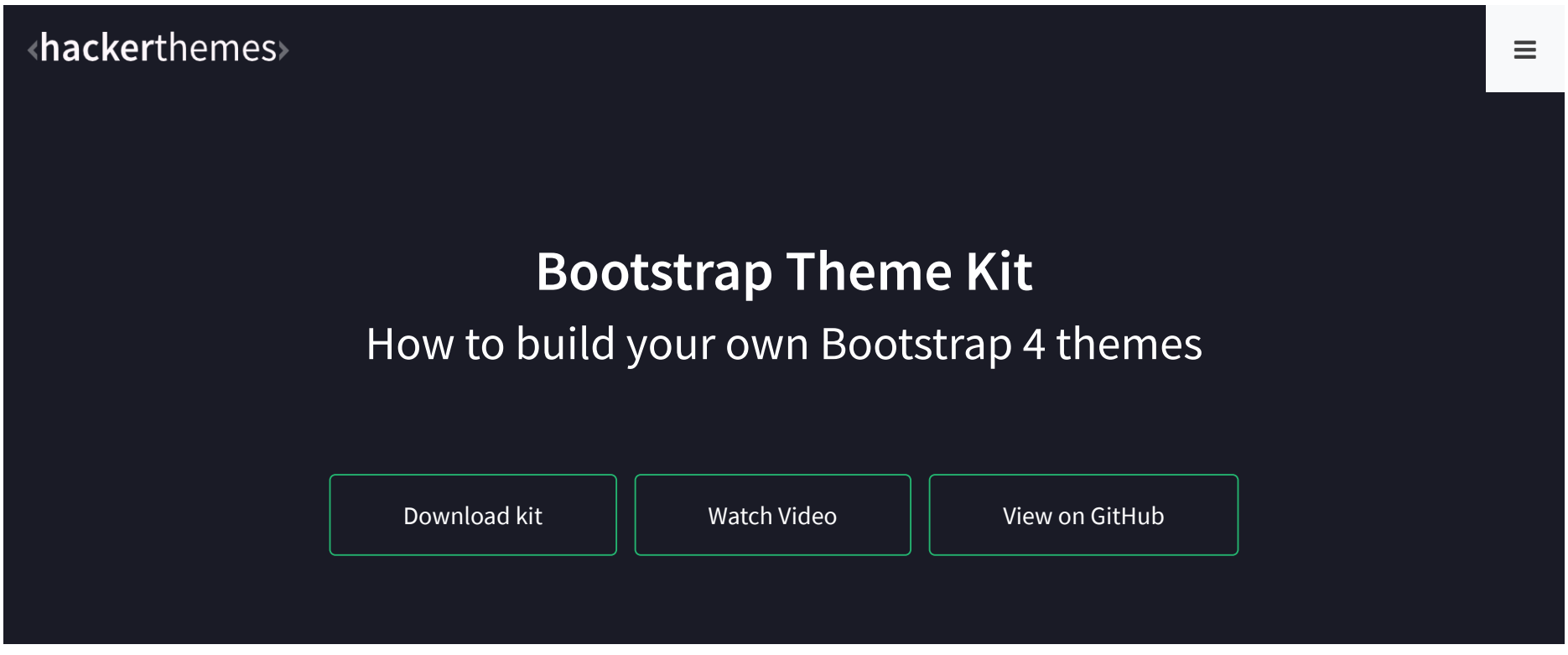


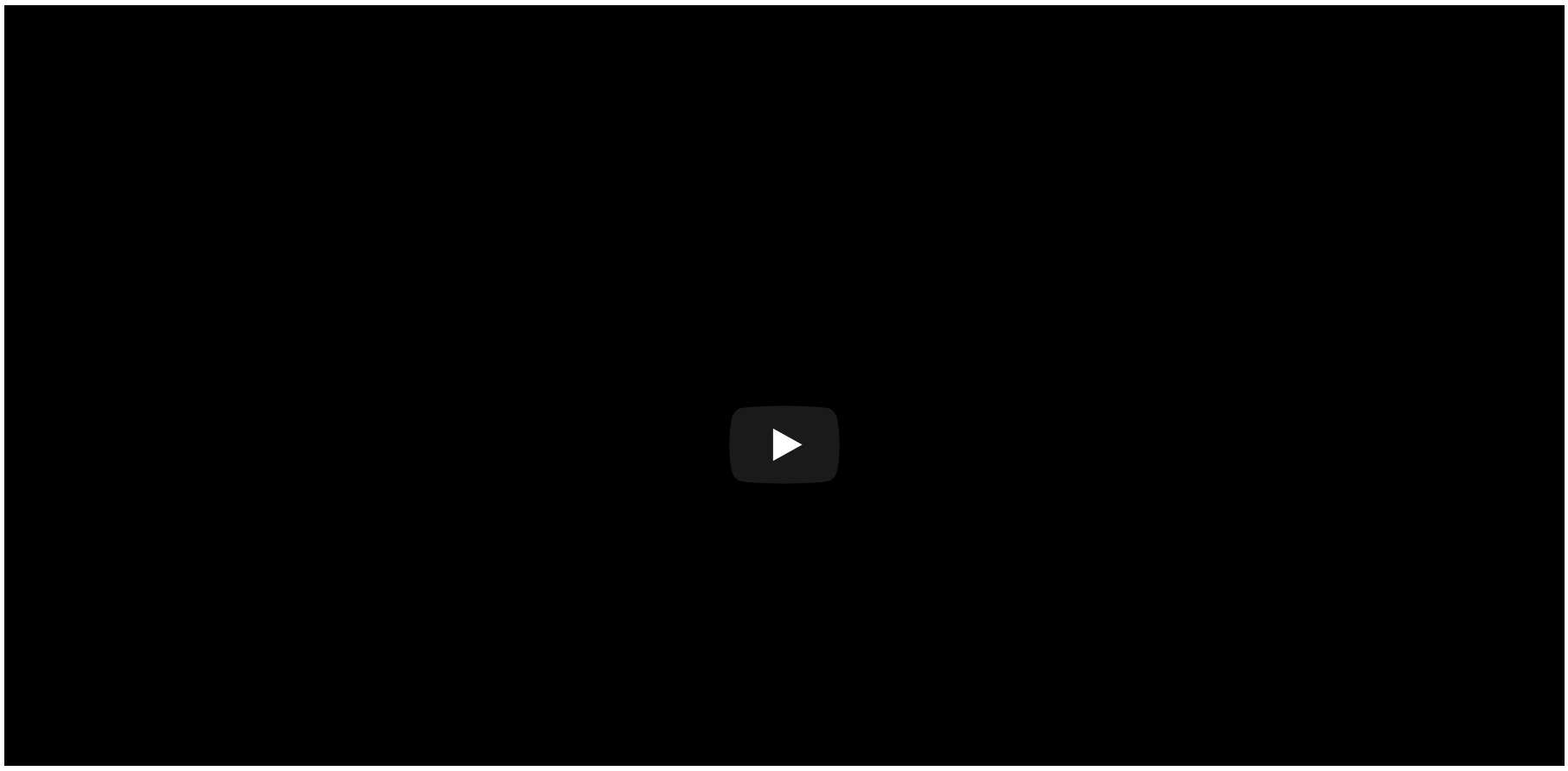

Share Tweet Star

## Introduction

In this tutorial I will show you how to create your own Bootstrap 4 themes using Sass, Gulp and the Bootstrap source code. The result will be a theme that you can apply to new and existing Bootstrap 4 projects to give them a new and fresh look. I've already made some open source themes with this workflow, check out Neon Glow or Vibrant Sea for to see two examples. If you prefer a less code-intense approach, you can also check out this Bootstrap builder which offers a web interface to customize the default Bootstrap 4 theme.

## Demonstration

In this video I'll show you how the Theme Kit works. You can watch this for a quick view of the essentials. If you're really in a hurry you can even skip the intro and start at 1:34 for the most juicy part. Farther below you'll find step-by-step instructions to get started.



## Step 1: Prerequisites

*Short version:* You need npm and gulp. A text editor and your command line too.

*Long version:* In order to build our own theme, we will rebuild Bootstrap 4 from the source code, modifying Sass variables and making use of Bootstrap's mixins and functions. To use the theme kit you'll need at least a text editor and node.js installed on your system. You can get node from nodejs.org. Once installed, you should be able to run the command `npm` from your command line like so:

```
alex@ubuntuVM:~$ npm -v
5.5.1
```

After that, you can install a node package called Gulp, which you can do by typing `npm install -g gulp-cli`. Finally you should be able to run `gulp`:

```
alex@ubuntuVM:~$ gulp -v
[16:35:47] CLI version 1.4.0
```

*Note:* Your version numbers could be slightly different, but that shouldn't be a problem.

## The Bootstrap Theme Kit

Despite its glamorous name and pompous presentation, the theme kit is a humble package. It's a basic project that includes three main parts:

- A Sass file structure with the correct imports already in place
- A Gulp script that takes care of building, minifying and prefixing the CSS code
- An HTML file with Bootstrap components to allow for a quick feedback loop

If you've already got a web project, you would probably want to imitate these techniques with your own build system. The theme kit is simply a minimal project that allows for a quick start and to play around with the theme features without too much effort.

## Step 2: Building the theme

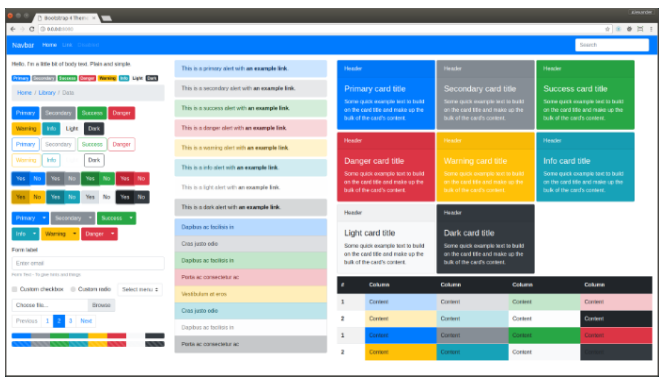
After downloading and extracting the Theme Kit, you'll have to run `npm install` inside the directory to pull all the dependencies. This will download the Bootstrap 4 source code as well as the Sass transpiler for you.

```
alex@ubuntuVM:~/theme-kit$ npm install
...
added 499 packages in 5.839s
```

Oof, that's a lot of packages. Welcome to npm :-). The packages are stored under *node_modules/,* so they won't pollute anything outside the project directory. Once the dependencies are installed, you can run `gulp` to run the build script. You can also run `gulp watch` to run the Gulp watcher which will run the build each time you change one of the Sass files. This is the most practical thing to do when creating a theme.

```
alex@ubuntuVM:~/theme-kit$ gulp watch
[18:02:30] Using gulpfile ~/theme-kit/gulpfile.js
[18:02:30] Starting 'build-theme'...
[18:02:32] Finished 'build-theme' after 1.45 s
[18:02:32] Starting 'watch'...
[18:02:32] Finished 'watch' after 5.39 ms
```

## Step 3: Displaying the theme in the browser



Now that we have the build script running, let's have a look at our theme's current state in the browser. You can open *index.html* in your browser of choice. *Optional*: What I usually do here is I run a simple development web server right in the project directory using something like `php -S 0.0.0.0:8080`, so I can then navigate to *localhost:8080* instead of opening the file from the file explorer. But this is not strictly necessary.

As you can see, there's a collection of relevant components on the screen, organized into a layout that fits a 1920x1080 display. At this point they're built with the default Bootstrap 4 style, but we're about to change that.

## Step 4: Modifying the variables

With the gulp watcher running in the background and our testing page loaded in the browser we can now get to theming. Let's have a look at the main Sass file now, which is *scss/mytheme.scss*:

scss/mytheme.scss

```
@import 'custom-variables';
@import '../node_modules/bootstrap/scss/bootstrap';
@import 'custom-styles';
```

As you can see, this imports our two other Sass files and the Bootstrap source code which is under *node_modules/bootstrap/*. Our files *custom-variables* and *custom-styles* are empty, which is why our theme is essentially just standard Bootstrap 4 right now.

Okay, let's start off with an easy win: Modifying the existing Bootstrap variables. Open *node_modules/bootstrap/scss/_variables.scss* to have a look at the options that Bootstrap gives us:

node_modules/bootstrap/scss/_variables.scss

```
// this is but a tiny snippet of _variables.scss
$primary:       $blue !default;
$secondary:     $gray-600 !default;
$success:       $green !default;
$info:          $cyan !default;
$warning:       $yellow !default;
$danger:        $red !default;
$light:         $gray-100 !default;
$dark:          $gray-800 !default;
```

To modify these, we'll make use of the `!default` rule. Notice how every last variable in this file has `!default` app to it. What this means is that the value will only be assigned if it **has not yet been defined earlier**. So to override these values we simply define them **earlier** in the Sass code.

This is the opposite of what you would expect when you think about overriding something in CSS. But it's a very powerful mechanic that allows for modifying the colors without injecting any files into Bootstrap's source. The files under *node_modules/* remain completely untouched. This is important enough to repeat: Don't change anything in the *node_modules/* directory unless you want to lose your work later.

If you take a look at our file *scss/mytheme.scss* above, you see that the file *scss/_custom-variables.scss* is included *before* Bootstrap. So we can use this file to override any of the default variables that Bootstrap uses. Let's add some variables to *scss/_custom-variables.scss* and see how our theme starts to take a different form:

scss/_custom-variables.scss

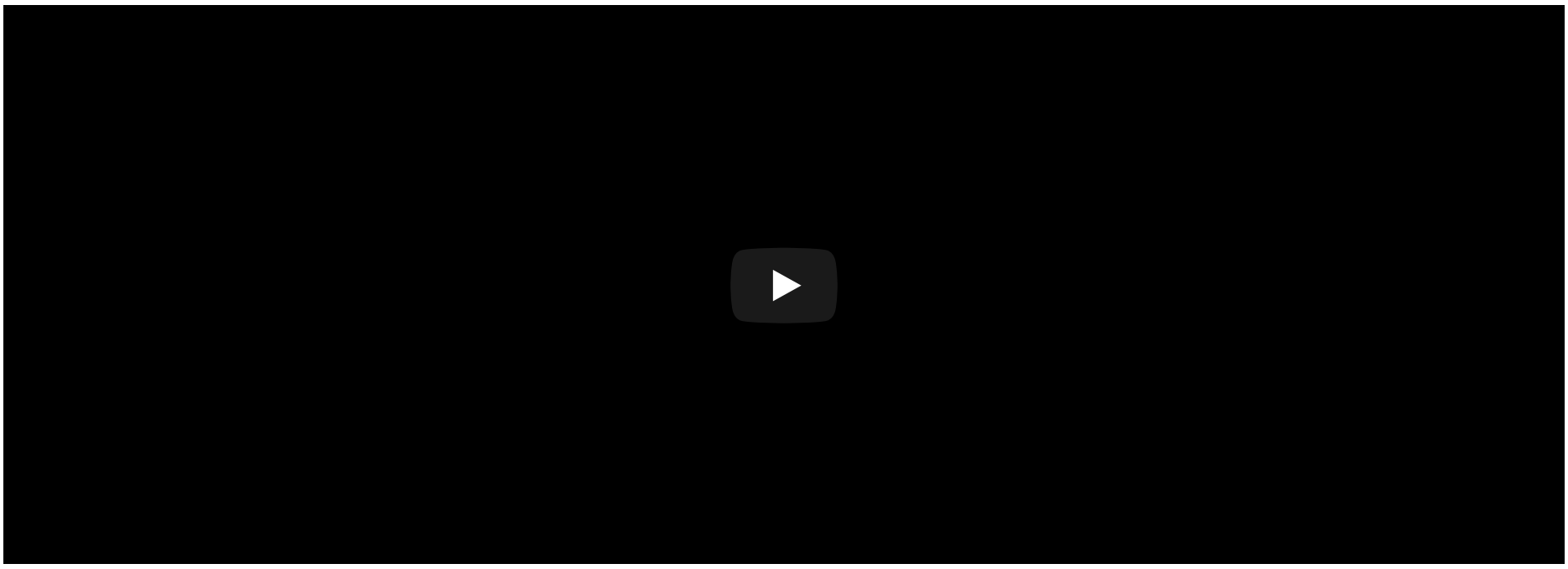
```
// Changing the theme colors
$primary: #3ec89d;
$secondary: #3ab7ff;
$success: #65ff9f;
$info: #7164ff;
$warning: #ff9f65;
$danger: #ff457b;
$light: #f2d4ff;
$dark: #18181d;
```

If you save this and refresh your browser, you'll see that all the components have gotten a paint job. At this point I should mention that I'm using a browser plugin for Chrome called *LivePage* which reloads the page automatically for me on changes. My code editor also has a nifty color picker for CSS colors.

Together with Gulp chugging along in the background, I don't even have to leave my code editor anymore. All of this combined, my workflow looks like this (this video is real time and without sound):

Modifying the variables in the code editor

Changes appear in the browser after each time I hit Ctrl+S



You gotta admit that's a pretty neat setup! :-) There are some variables that you can uncomment in *_custom-variables.scss* but I would encourage you to take a good look at Bootstrap's *_variables.scss* file to find out what

other values you can tinker with.

Some that are worth mentioning are `$border-radius`, `$enable-gradients`, `$enable-shadows`. The last two give Bootstrap back some of it's traditional look when set to true.

## Step 5: Extending maps to create new components

If you look at Bootstrap's variable file again, you'll see that there are these map structures:

node_modules/bootstrap/scss/_variables.scss

```
// this is but another tiny snippet of _variables.scss
$theme-colors: () !default;
$theme-colors: map-merge((
  "primary":    $primary,
  "secondary":  $secondary,
  "success":    $success,
  "info":       $info,
  "warning":    $warning,
  "danger":     $danger,
  "light":      $light,
  "dark":       $dark
), $theme-colors);
```

These are very interesting for us, because they're looped over later in the Bootstrap code to create all sorts of components. Some examples are `btn-primary`, `alert-secondary`, `bg-danger` and so on. We can amend these maps painlessly, thanks to the `!default` feature and the way these maps were built by the Bootstrap team. In the code snippet above, the map *$theme-colors* is defined as a default value, and then merged with the native Bootstrap colors. This is a very clever implementation because it allows us to extend the map simply by defining our own map in our *_custom-variables.scss* file:

scss/_custom-variables.scss

```
$theme-colors: (
 "cool": #2a3f81
);
```

With three very humble lines of code we're causing a whole set of new components to be generated. We can now use them in our HTML code like this:

index.html

```
<a href="#!" class="btn btn-cool">A cool button</a>
<div class="alert alert-cool">A cool alert</div>
```

The result looks like this:

## Step 6: Making use of Bootstrap mixins and functions

So far we've only modified the input variables for standard Bootstrap. This provides us with a solid foundation to work from, but there's more power to be leveraged. If we take a look at our main file *scss/mytheme.scss* again, you'll see there is another file included called *custom-styles*:

scss/mytheme.scss

```
@import 'custom-variables';
@import '../node_modules/bootstrap/scss/bootstrap';
@import 'custom-styles';
```

In this file you can put all of your own styles, including those that overwrite some of Bootstrap's styles in the traditional way. This is no different from including your own CSS later in the HTML code and and overwriting stuff as you need. This is likely what you've been doing until today, right?

The reason I'm including it here is this:

Because we included Bootstrap earlier in the same Sass file, we have at our disposal all of the mixins, functions and variables that are included in Bootstrap. These are used by the Bootstrap source code itself to generate components and we can use them to build our own custom components with total control.

Let's get on with an example. Have a look at the folder *node_modules/bootstrap/scss/mixins*. There are a bunch of files that each define sepearte mixins related to certain components. For our example we'll use the one from *_alerts.scss*.

node_modules/bootstrap/scss/mixins/_alert.scss

```
@mixin alert-variant($background, $border, $color) {
  color: $color;
  @include gradient-bg($background);
  border-color: $border;

  hr {
    border-top-color: darken($border, 5%);
  }

  .alert-link {
    color: darken($color, 10%);
  }
}
```

As you can see there is no magic involved. We can simply call this mixin in our file *_custom-styles.scss* like this:

scss/_custom-styles.scss

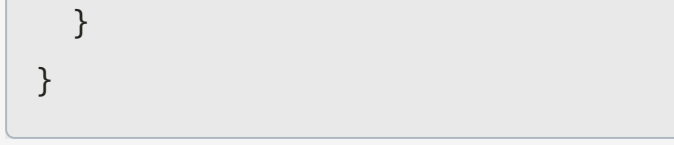
```
.alert-myalert {
  @include alert-variant(#60667d, #1d1d1d, #f4fdff);
}
```

And use it in our HTML code like so:

index.html

```
<div class="alert alert-myalert">My own alert</div>
```

Because we can define all three colors (background, border and text) we have more control over the component. Of course we could add any styles we want right below the @include line to fully customize the alert. This is how this alert looks. Again, 3 lines of Sass code:




## Step 7: Looping over the $theme-colors map

Let me pull one last stunt. Let's say we wanted to change the way the default alerts look and give them all a solid shadow that corresponds to their color. We could of course overwrite them one by one. But remember, we have Bootstrap's variables available to us in *custom-styles* and that includes the `$theme-colors` map.

Let's look at how alert variants are generated by Bootstrap in the first place:

node_modules/bootstrap/scss/_alerts.scss

```
// Generate contextual modifier classes for colorizing the alert.

@each $color, $value in $theme-colors {
  .alert-#{$color} {
    @include alert-variant(theme-color-level($color, -10),
                           theme-color-level($color, -9),
                           theme-color-level($color, 6));
  }
}
```

The `$theme-colors` map is looped over and for each color, the `alert-variant` mixin is called. There's also a function called `theme-color-level` which weakens or darkens a color by a numerical value. This is a Bootstrap function, not a native Sass function, by the way.

So let's use this pattern to amend those alerts in our own code, in *scss/_custom-styles.scss*:

scss/_custom-styles.scss

```
@each $color, $value in $theme-colors {
  .alert-#{$color} {
    box-shadow: 3px 3px theme-color-level($color, -3);
  }
}
```

For each alert we'll add a rule that adds a box shadow with the correct color, softened up a little bit. With everything else on default, they look like this:


If you've modified the theme colors earlier, the shadows will look different of course. Which is exactly why I like using this technique. The shadows respond to the theme colors just like everything else.

## Step 9: Enjoy your new look

Okay, our basic Bootstrap theme is now officially done. Of course there is much more you can do. But these are some basic techniques that give you a good foundation to build upon. You can now take the file *css/mytheme.min.css* and use it instead of Bootstrap. It essentially still *is* Bootstrap, so it works exactly the same.

Check out Bootstrap's *_variables.scss* file, the *scss/_mixins* folder and the *_functions.scss* file for more juicy mixins and functions (they're all under *node_modules/bootstrap/scss*).

There is an official documentation on theming for more info on the topic.

Also check out the open source Bootstrap 4 themes on the front page for some inspiration. If you choose to build and publish your own theme, let me know via twitter or email.

Share Tweet Star

‹hackerthemes›

Terms and Conditions Privacy Policy
About Support Blog