Creating Your Own Build Process with Gulp

Before you start with Lesson 8, I will first show you how to set up your own build process for your Bootstrap project.

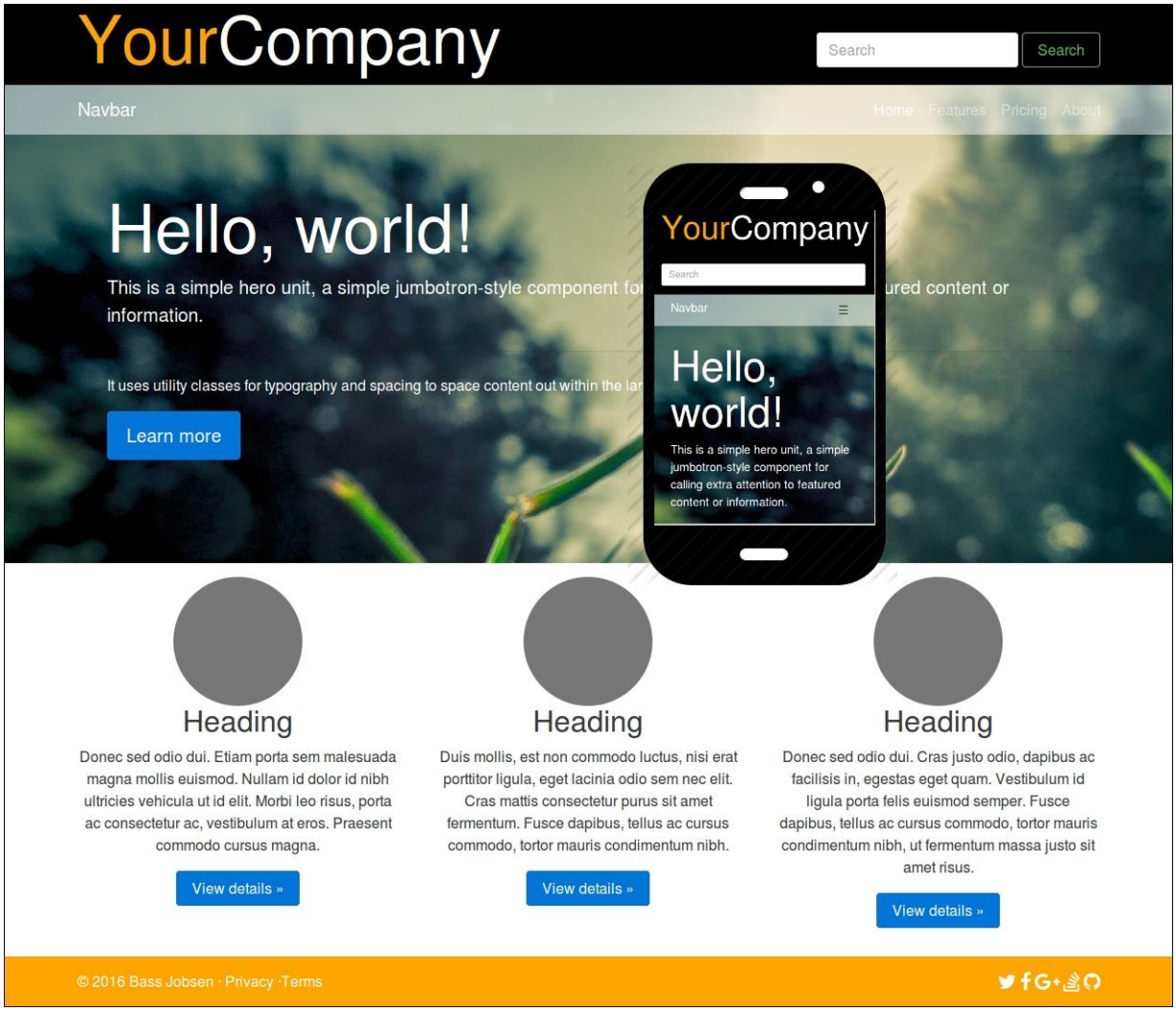
After reading this, you will have learned to:

- Set up a build process with Gulp
- Create different environments in your build processor
- Compile your Sass code into CSS
- Automatically add vendor prefixes to your CSS code
- Prepare the JavaScript plugin for your project
- Run a static web server
- Test your code
- Use standard Bootstrap components and tweak them to fit your needs
- Create a simple one-page marketing website with Bootstrap
- Publish your project on GitHub

What are we going to build?

You will create a build process for an example project. The process not only prepares your CSS and JavaScript code, but also runs a static web server, which automatically reloads on file changes in browser testing, runs some tests for your code, and more.

To demonstrate the different steps of the build process, you will create an example project. The example project is a simple one-page marketing layout powered by Bootstrap. At the end, you should have created an HTML page, which will look like that shown in the following screenshot:



Requirements

The first requirement is of Node.js that you've already installed in your system. After installing Node.js and `npm`, you should also install Gulp globally on your system. You can install Gulp by running the following command in your console:

```
npm install --global gulp-cli
```

Gulp is a task runner for Node.js. You can use it to create a build system. You can easily add automating tasks such as modification and copying to the build system. You can also use Gulp to create watch events. These events automatically rerun the tasks of the build system when a file changes. Bootstrap's CSS code is built with Sass, and you can use Gulp to compile your Sass code into static CSS code. After compiling your Sass code, you can build a task to post-process your CSS code.

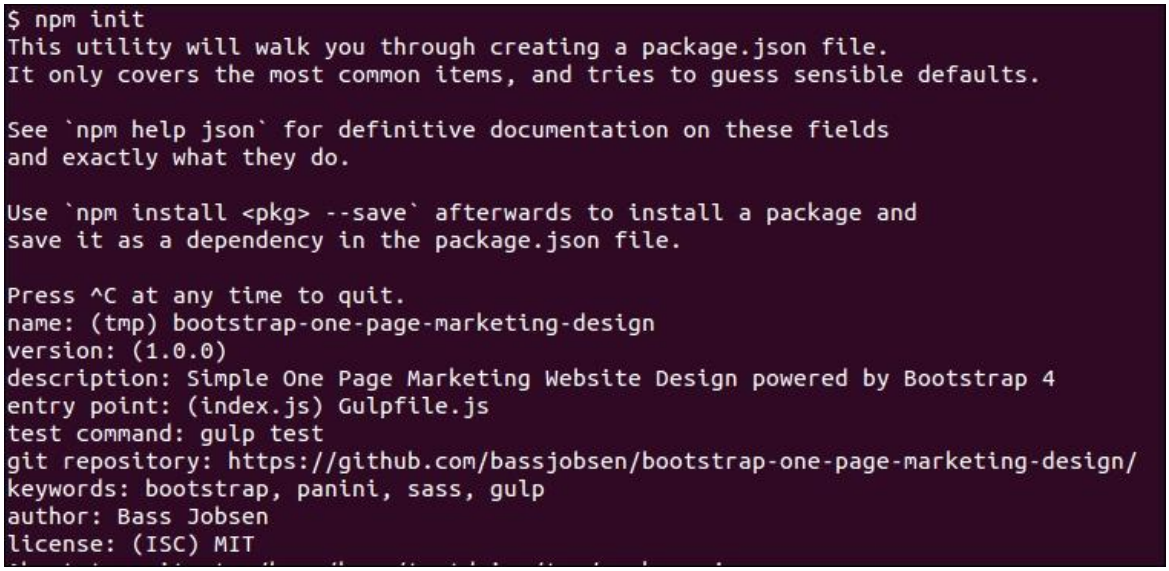
Further on you will read that you can also use Gulp to test your code, run some linters, and get your code ready for production by minifying and optimizing it. Linting is the process of running a program that will analyze code for potential errors.

Installing Gulp in your project

Before we go any further, we must first initiate `npm` for our project by running the following command in our console:

```
npm init
```

Answer the questions like those shown in the following screenshot:



The preceding command creates a new `package.json` file; this file holds various metadata relevant to the project, and it also contains the dependencies for the project. The Gulp plugins you will install later on are all (dev) dependencies of your project.

Now install Gulp in your project, `devDependencies`, by running the following command in your console:

```
npm install --save-dev gulp
```

The `--save-dev flag` writes down Gulp as a devDependency into the `package.json` file.

Creating the Gulpfile.js which holds your tasks

Create a new file called `Gulpfile.js` in your project directory and write down the following JavaScript code into it:

```
var gulp = require('gulp');

gulp.task('default', function() {
  // place code for your default task here
});
```

Later on you will add your Gulp task into this file. You can test it already by running the following command in your console:

```
gulp
```

The default task will run and do nothing.

Now you're ready to install not only Bootstrap but also the required Gulp plugins for your build process.

The clean task

Each time the build process runs, the clean task should remove the temporary `_site` directory and all of its contents:

```
// Erases the dist folder
gulp.task('clean', function() {
  rimraf('_site');
});
```

Notice that you'll have to install `rimraf` as follows:

```
npm install rimraf --save-dev
```

Setting up environments for development and production

Of course, you will need your build process to develop your project, but after that you will run different tasks to get your code ready for production. In the development stage, you need CSS code with **CSS sourcemaps** for easy debugging, and in the production stage, you probably will minify the CSS code without CSS sourcemaps.

The `gulp-environments` plugin makes it convenient to create separate environments, such as development and production, to run your tasks in. You can install this plugin by running the following command:

```
npm install --save-dev gulp-environments
```

Then add the plugin to your `Gulpfile.js` file, which should look like this:

```
var environments = require('gulp-environments');
```

Then assign the environments as follows:

```
var development = environments.development;
var production = environments.production;
```

Now you can pass a command line flag `--env` to set the environment:

```
gulp build --env development
```

You can also, conditionally assign a variable like:

```
var source = production() ? "source.min.js" : "source.js";
```

Or run sub tasks for only one environment:

```
.pipe(development(sourcemaps.init()))
```

> **Note**
>
> The full documentation and some examples of the gulp-environments plugin can be found at: https://github.com/gunpowderlabs/gulp-environments.

Installing Bootstrap via Bower

Bower is a package management system for client-side programming. You can use Bower to install Bootstrap's source code and keep it up to date more easily. Initiate Bower by running the following command in your console:

```
bower init
```

Answer the questions like those shown in the following screenshot:

```
$ bower init
? name bootstrap-one-page-marketing-design
? description Simple One Page Marketing Website Design powered by Bootstrap 4
? main file Gulpfile.js
? what types of modules does this package expose?
? keywords bootstrap, panini, gulp, sass
? authors Bass Jobsen <bass@w3masters.nl>
? license MIT
? homepage https://github.com/bassjobsen/bootstrap-one-page-marketing-design/
? set currently installed components as dependencies? No
? add commonly ignored files to ignore list? Yes
? would you like to mark this package as private which prevents it from being ac
cidentally published to the registry? Yes
```

After the preceding steps, a `bower.json` file is created. Then run the following command in your console:

```
bower install bootstrap#4 --save-dev
```

The preceding command downloads Bootstrap to the `bower_components` folder. Notice that jQuery and Tether are also installed since Bootstrap depends on these libraries.

The `--save-dev` flag writes down Bootstrap as a `devDependency` into the `bower.json` file.

Creating a local Sass structure

Before we can start compiling Bootstrap's Sass code into CSS code, we have to create some local Sass or SCSS files. First, create a new `scss` subdirectory in your project directory. In the `scss` directory, create your main project file called `app.scss`.

Then create a new subdirectory in the new `scss` directory called `includes`. Now you will have to copy the `bootstrap.scss` and `_variables.scss` from the Bootstrap source code in the `bower_components` directory to the new `scss/includes` directory:

```
cp bower_components/bootstrap/scss/bootstrap.scss
scss/includes/_bootstrap.scss
cp bower_components/bootstrap/scss/_variables.scss scss/includes/
```

Notice that the `bootstrap.scss` file has been renamed as `_bootstrap.scss`, starting with an underscore, and has become a partial file now.

Import the files you have copied in the previous step into the `app.scss` file as follows:

```
@import "includes/variables";
@import "includes/bootstrap";
```

Then open the `scss/includes/_bootstrap.scss` file and change the import part for the Bootstrap partial files, so that the original code in the `bower_components` directory will be imported here. Notice that we set the include path for the Sass compiler to the `bower_components` directory later on. The `@import` statements should look like that shown in the following SCSS code:

```
// Core variables and mixins
@import "bootstrap/scss/variables";
@import "bootstrap/scss/mixins";
// Reset and dependencies
@import "bootstrap/scss/normalize";
.....
```

You're importing all of Bootstrap's SCSS code in your project now. When preparing your code for production, you can consider commenting out the partials you do not require for your project.

Modification of the `scss/includes/_variables.scss` is not required, but you can consider removing the `!default` declarations because the real default values are set in the original `_variables.scss` file which is imported after the local one.

Notice that the local `scss/includes/_variables.scss` file does not have to contain a copy of all Bootstrap's variables. Having them all makes it easier to modify them for customization; it also ensures that your default values do not change when updating Bootstrap.

Compiling Bootstrap's Sass code into CSS code

Now it's time to start compiling the Sass code into CSS code. Bootstrap's Sass code is written in the newer SCSS syntax. Two Gulp plugins are available to compile Sass into CSS with Gulp. The first plugin is called `gulp-ruby-sass` and as its name already tells you, it compiles Sass to CSS with Ruby Sass. The second `gulp-sass` plugin uses `node-sass` to compile your Sass code with `libSass`. In this course, a `gulp-sass` plugin is used. Notice that Compass is not compatible with `libSass`.

You can install `gulp-sass` in your project by running the following command in your console:

```
npm install gulp-sass --save-dev
```

After installing the plugin, you can add the compile task to your `Gulpfile.js` as follows:

```
var sass = require('gulp-sass');
var bowerpath = process.env.BOWER_PATH || 'bower_components/';
var sassOptions = {
  errLogToConsole: true,
  outputStyle: 'expanded',
  includePaths: bowerpath
};

gulp.task('compile-sass', function () {
    return gulp.src('./scss/app.scss')
        .pipe(sass(sassOptions).on('error', sass.logError))
        .pipe(gulp.dest('./_site/css/'));
});
```

Notice that we set the `includePaths` option to the `bowerpath` variable as described previously; and also the `development()`condition is used to write CSS sourcemaps only in the development stage. You can read more about CSS sourcemaps in the next section.

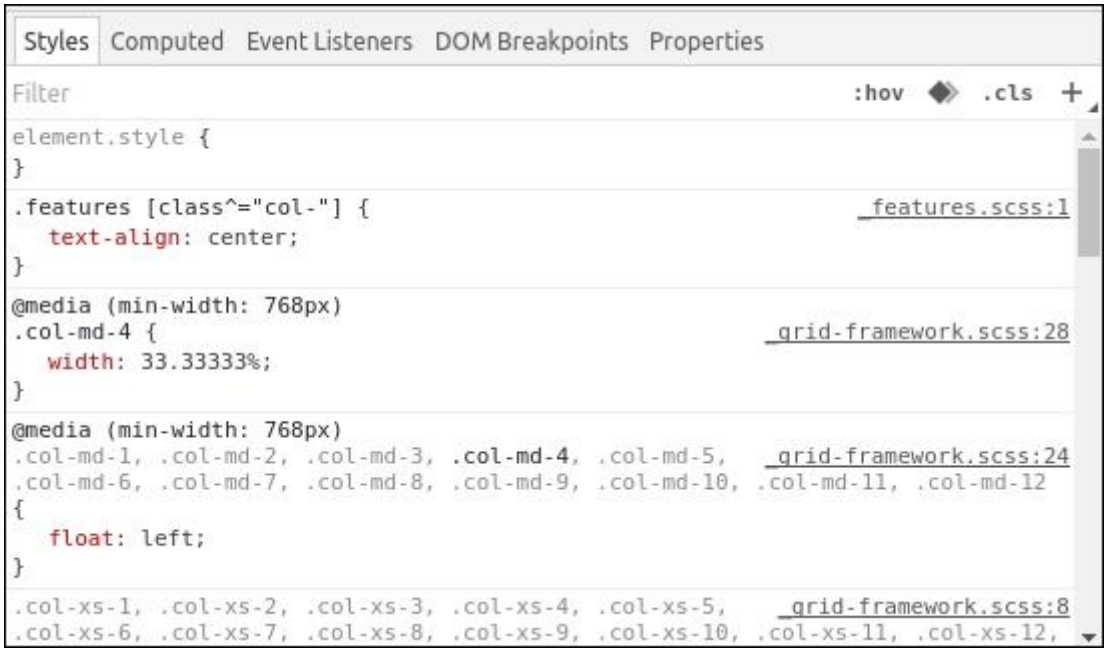
CSS sourcemaps for debugging

The Sass compiler merges different Sass files to a single CSS file. This CSS file has been minified in most cases. When you are inspecting the source of HTML files with the developer tools of your browser, you cannot relate the style effects to your original Sass code. CSS Sourcemaps solve this problem by mapping the combined/minified file back to its unbuilt state.

CSS source maps were introduced to map minified JavaScript to its origin source. Since version 3 of the CSS source map protocol, it has also support for CSS. The Sass compiler (or better, the `gulp-sourcemap` plugin in this case) generates the CSS source map and adds a reference into the CSS file as follows:

```
/*# sourceMappingURL=app.css.map */
```

In the following screenshot, you will find an example of the style rules pointing to their Sass file of origin in the browser's developer's tools:



To use CSS sourcemaps, you have to install the `gulp-sourcemaps` plugin first:

```
npm install gulp-sourcemaps --save-dev
```

After that you can add the sourcemaps to your `compile-sass` task as follows:

```
gulp.task('compile-sass', function () {
    return gulp.src('./scss/app.scss')
        .pipe(development(sourcemaps.init()))
        .pipe(sass(sassOptions).on('error', sass.logError))
        .pipe(development(sourcemaps.write()))
        .pipe(gulp.dest('./_site/css/'));
});
```

Notice that the CSS sourcemaps are only generated in the development stage due to the `development()`wrapping and the `gulp-environments` plugin.

The `gulp-sourcemaps` plugin does also support the `gulp-postcss` and `gulp-cssnano` plugins as described later on. You should call these plugins between the `sourcemaps.init()` and `sourcemaps.write()`calls in the `compile-sass` task.

Running the postCSS autoprefixer

Since version 3.2, Bootstrap requires the autoprefixer plugin for vendor prefixing your CSS code. Running the postCSS autoprefixer plugin is easy and fully automated with Gulp.

The `gulp-postcss` plugin enables you to run other plugins to post-process your compiled CSS code.

Bootstrap requires the `postcss` autoprefixer which you can run with the `gulp-postcss` plugin. The autoprefixer automatically adds the required vendor prefixes to your CSS code, and it uses the data of the Can I Use database (`http://caniuse.com/`).

Notice that Bootstrap also requires the `mq4-hover-shim`, which is a shim for the Media Queries Level 4 hover @media feature.

See `https://www.npmjs.com/package/mq4-hover-shim` for more details about this shim.

Perform the following steps to install and configure both the `autoprefixer` and `mq4-hover-shim`. Start by running the following command in your console:

```
npm install gulp-postcss autoprefixer mq4-hover-shim --save-dev
```

Now you can declare a variable which contains your processors for the `gulp-postcss` plugin at the `Gulpfile.js`:

```
    var processors = [
    mq4HoverShim.postprocessorFor({ hoverSelectorPrefix: '.bs-true-hover ' }),
    autoprefixer({
   browsers: [
    //
    // Official browser support policy:
    // http://v4-alpha.getbootstrap.com/getting-started/browsers-
   devices/#supported-browsers
    //
    'Chrome >= 35',
    'Firefox >= 38',
    'Edge >= 12',
    'Explorer >= 9',
    'iOS >= 8',
    'Safari >= 8',
    'Android 2.3',
    'Android >= 4',
    'Opera >= 12'
    ]
    })
```

```
    ];
```

As you can see, the autoprefixer accepts a hash argument which sets which browser should be supported. You can copy this hash from the `bower_components/bootstrap/Grunt.js` file of Bootstrap's source files.

After setting your processors you can add the postcss plugin to your sass compile task as follows. Notice also that the `gulp-postcss` plugin supports the `gulp-sourcemaps` plugin:

```
gulp.task('compile-sass', function () {
    return gulp.src('./scss/app.scss')
        .pipe(development(sourcemaps.init()))
        .pipe(sass(sassOptions).on('error', sass.logError))
        .pipe(postcss(processors))
        .pipe(development(sourcemaps.write()))
        .pipe(gulp.dest('./_site/css/'));
});
```

Getting your CSS code ready for production

In the production stage of your project, you do not need CSS sourcemaps anymore. The `development()` wrapping in the `compile-sass` task already guarantees that CSS sourcemaps are not generated when the environment is set to production by the `gulp-environments` plugin.

The smaller your compiled CSS code, the faster it will load in your browser. So minifying the compiled CSS code will allow faster loading. CSSnano runs your CSS code through many focused optimizations, to ensure that the final result is as small as possible for a production environment.

A Gulp plugin for CSS nano is available too; you can install it by running the following command in your console:

```
npm install gulp-cssnano --save-dev
```

After installing the plugin, you can add it to your `compile-sass` task:

```
var cssnano = require('gulp-cssnano');
gulp.task('compile-sass', function () {
    return gulp.src('./scss/app.scss')
        .pipe(development(sourcemaps.init()))
        .pipe(sass(sassOptions).on('error', sass.logError))
        .pipe(postcss(processors))
        .pipe(production(cssnano()))
        .pipe(development(sourcemaps.write()))
        .pipe(gulp.dest('./_site/css/'));
});
```

The preceding code makes it clear that `cssnano` only runs when the environment is set to production. You can also run `cssnano` as a processor for the `gulp-postcss` plugin.

The most common issue for CSS performance is unused CSS code. You can comment out the CSS components and other code that you do not use in the `scss/includes/_bootstrap.scss` file to make your compiled CSS code smaller.

Linting your SCSS code

When you are using Sass to extend and modify Bootstrap's CSS code, it is important that you keep your code clean and readable. The SCSS-lint tool can help you to write clean and reusable SCSS code. A Gulp plugin is available for the SCSS-lint tool and you can install it by running the following command in your console:

```
npm install gulp-scss-lint --save-dev
```

Notice that the `gulp-scss-lint` plugin requires both Ruby and SCSS-lint installed already.

Now you can add the `scss-lint` task to your `Gulpfile.js`:

```
gulp.task('scss-lint', function() {
  return gulp.src('scss/**/*.scss')
    .pipe(scsslint());
});
```

Bootstrap has its own configuration file for the SCSS-lint tool. You can reuse this configuration for your custom code, and copy the configuration file from the Bootstrap source code to your local `scss` directory:

```
cp bower_components/bootstrap/scss/.scss-lint.yml scss/
```

Then modify your scss-lint task so that the configuration file is used:

```
gulp.task('scss-lint', function() {
   return gulp.src('scss/**/*.scss')
      .pipe(scsslint({'config': 'scss/.scss-lint.yml'}));
});
```

When running the linter over the current Sass files in the `scss` directory, include the local copies of the `_variable.scss` and `_bootstrap.scss` file. You will find that the comments in the `_bootstrap.scss` will give a warning.

In the `.scss-lint.yml`, you can enable this warning by changing the comments setting as follows:

```
Comment:
   enabled: true
   exclude: ['_bootstrap.scss']
```

Also in the example code, the following configuration has been changed:

```
ColorKeyword:
   enabled: false
```

Using keywords for colors, for instance orange instead of `#ffa500` is mostly considered as a bad practice. In the example code in this course, color names are preferred because they're easier to read by humans.

Of course, you can add the `scss-lint` task to your default task, but you can also run it manually when needed. When adding the `scss-lint` task in the default task, you should consider using it together with the `gulp-cached` plugin to only lint the modified files:

```
var scsslint = require('gulp-scss-lint');
var cache = require('gulp-cached');
gulp.task('scss-lint', function() {
  return gulp.src('/scss/**/*.scss')
    .pipe(cache('scsslint'))
    .pipe(scsslint());
});
```

You should also use the `gulp-cached` plugin as shown in the preceding code when linting your files via the watch task as described further on here.

Preparing the JavaScript plugins

Some of Bootstrap's components do not only require CSS but also JavaScript. Bootstrap comes with jQuery plugins for commonly used components. Similarly, the navbar requires the collapse plugin and the carousel component has its own plugin too.

Bootstrap's plugins require jQuery and the Tooltips and Popover components also require the Tether library.

In the build process, you can bundle jQuery, Tether, and the plugins into a single file by using the `gulp-concat` plugin.

You can install the `gulp-concat` plugin by running the following command in your console:

```
npm install gulp-concat --save-dev
```

After that the task that bundles the JavaScript files may look like:

```
gulp.task('compile-js', function() {
  return gulp.src([bowerpath+ 'jquery/dist/jquery.min.js', bowerpath+
'tether/dist/js/tether.min.js', bowerpath+
'bootstrap/dist/js/bootstrap.min.js','js/main.js'])
    .pipe(concat('app.js'))
    .pipe(gulp.dest('./_site/js/'));
});
```

The preceding `compile-js` task also bundles a local `js/main.js` file; you can put the plugin settings or your custom JavaScript code into this file.

Getting your JavaScript code ready for production

The preceding `compile-js` task only concatenates the JavaScript files which are already compiled and minified. These files are loaded from the `bower_components` directory.

The `bootstrap.min.js` file contains all the plugins. You will probably only use some of these plugins.

To create a smaller JavaScript file, you can only bundle the plugins you need and minify the code yourself in the build process.

You can minify your code with the `gulp-uglify` plugin that can be installed with the following command:

```
npm install --save-dev gulp-uglify
```

Your `compile-js` task may look like the following when using the `gulp-uglify` plugin:

```
gulp.task('compile-js', ['compress']);
```

After that the task that bundles the JavaScript files may look like this:

```
var uglify = require('gulp-uglify');

gulp.task('compress', function() {
  return gulp.src([
    bowerpath+ 'jquery/dist/jquery.js',
    bowerpath+ 'tether/dist/js/tether.js',
    bowerpath+ 'bootstrap/js/src/alert.js',
    bowerpath+ 'bootstrap/js/src/button.js',
    bowerpath+ 'bootstrap/js/src/carousel.js',
    bowerpath+ 'bootstrap/js/src/collapse.js',
    bowerpath+ 'bootstrap/js/src/dropdown.js',
    bowerpath+ 'bootstrap/js/src/modal.js',
    bowerpath+ 'bootstrap/js/src/popover.js',
    bowerpath+ 'bootstrap/js/src/scrollspy.js',
    bowerpath+ 'bootstrap/js/src/tab.js',
    bowerpath+ 'bootstrap/js/src/tooltip.js',
    bowerpath+ 'bootstrap/js/src/util.js',
    'js/main.js' // custom JavaScript code
    ])
    .pipe(uglify())
    .pipe(gulp.dest('dist/js/app.js'));
});
```

In the preceding code, leave out the JavaScript files you do not need for your project.

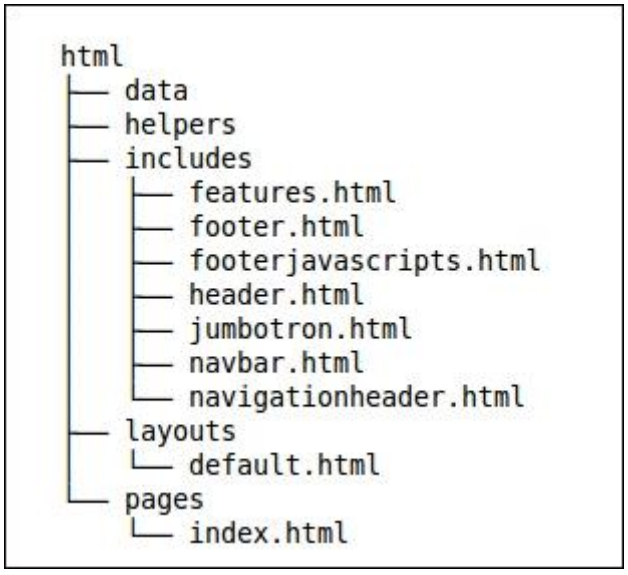
The preceding task also bundles the jQuery and Tether libraries, and you can also load these libraries via CDN. For this project, you can do this by using the following HTML in the `html/includes/footerjavascripts.html` template:

```
        <!-- jQuery first, then Bootstrap JS. -->
        <script
    src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/jquery.min.js"></sc
    ript>
        <script
    src="https://cdn.rawgit.com/HubSpot/tether/v1.2.0/dist/js/tether.min.js"></
    script>
        <script src="{{ root }}js/app.js"></script>
```

Modularization of your HTML

Although we'll only create a one-page design here, using a template engine for your project is not a bad idea. Whenever you decide to extend your project with one or more other HTML pages, DRY coding your HTML will make your work more efficient and reusable. Many HTML template languages and engines are available. Bootstrap uses the **Jekyll** static site generator to enable you to run its documentation locally. Many templates for Bootstrap CLI use Nunjucks now. Here, you will meet Panini for Gulp. Panini is a flat file compiler that uses the concepts of templates, pages, and partials powered by the Handlebars templating language.

For the marketing example project, you'll have to create the following file and directory structure, which holds your HTML templates:



In the preceding file and directory structure, the `pages` directory contains your pages. Each page has a layout, which can be found in the `layouts` directory. Both pages and layouts may include the HTML partials from the `includes` directory.

You can read more about Panini at `http://foundation.zurb.com/sitesdocs/panini.html` and `https://github.com/zurb/panini/`.

Installing the Gulp task to compile the Panini HTML templates

Run the following command to install the Panini compiler in your project:

```
npm install panini --save-dev
```

After installing the Panini compiler, add your HTML compile task to your `Gulpfile.js` file as follows:

```
var panini = require('panini');
gulp.task('compile-html', function(cb) {
  gulp.src('html/pages/**/*.html')
    .pipe(panini({
      root: 'html/pages/',
      layouts: 'html/layouts/',
      partials: 'html/includes/',
      helpers: 'html/helpers/',
      data: 'html/data/'
    }))
    .pipe(gulp.dest('_site'));
    cb();
});
```

Now you can run the `Gulp compile-html` command in your console.

The `compile-html` task saves the compiled HTML templates to the `_site` directory. The CSS and JavaScript tasks also save their products into this directory.

Validating the compiled HTML code

Bootlint is a tool that checks for several common HTML mistakes in web pages which are built with Bootstrap. Bootstrap's components and widgets require their parts of the DOM to conform to certain structures. Bootlint checks the HTML structure of your pages and ensures that it is correct for these components. Bootlint also checks if the HTML document contains the required meta tags.

Notice that Bootlint requires a valid HTML5 page in the first place. To ensure your code is valid HTML we'll run another linter before we run Bootlint. Run the following command in your console to install the `gulp-html plugin`:

```
npm install gulp-html --save-dev
```

`gulp-html` is a Gulp plugin for HTML validation, using `vnu.jar`.

Then install the `gulp-bootlint` plugin:

```
npm install gulp-bootlint --save-dev
```

You can easily initiate a `html-validate` task now, as follows:

```
var validator = require('gulp-html');
var bootlint  = require('gulp-bootlint');
gulp.task('validate-html', [compile-html], function() {
  gulp.src('_site/**/*.html')
    .pipe(validator())
    .pipe(bootlint());
});
```

Then the `validate-html` task should run after the `compile-html` task has finished. That's why the second parameter of the `validate-html` task is set to `[compile-html]`.

Read more about asynchronous task support in Gulp at `https://github.com/gulpjs/gulp/blob/master/docs/API.md#async-task-support`.

You should choose whether or not you run the validators in your build process by yourself. You can easily remove the `validate-html` task from the HTML task:

```
gulp.task('html', ['compile-html','validate-html']);
```

Creating a static web server

Now that your tasks to compile your HTML, CSS, and JavaScript code are ready, it's time to show and inspect the result in your browser. **Browsersync** is a module that keeps your browser in sync

when developing your code. Browsersync works by injecting an asynchronous script tag right after the `<body>` tag during initial request.

To use Browsersync with Gulp no special plugin is required; you can simply `require()` the module.

First install Browsersync by running the following command:

```
npm install browser-sync gulp --save-dev
```

Then create a task in your `Gulpfile.js` file which may look like the following:

```
var browser = require('browser-sync');
var port = process.env.SERVER_PORT || 8080;
// Starts a BrowerSync instance
gulp.task('server', ['build'], function(){  browser.init({server:
'./_site', port: port});});
```

The server task depends on the build task (second argument `['build']` in the preceding code) which means that the build task should run before the server task.

The server task starts a static web server running on port 8080 serving the files in the temporary `_site` directory. The compiled files from the other tasks are saved in the `_site` directory.

Start watching your file changes

When your static webserver is up and running you can add a watch task to your build process. The watch task should trigger a task on file changes and reload the web browser.

Your watch task may look like the following:

```
// Watch files for changes
gulp.task('watch', function() {
  gulp.watch('scss/**/*', ['compile-sass', browser.reload]);
  gulp.watch('html/pages/**/*', ['compile-html']);
  gulp.watch(['html/{layouts,includes,helpers,data}/**/*'], ['compile-
html:reset','compile-html']);
});
```

The watch task watches your Sass files in the `scss` folder and your HTML templates in the `html` folder. The `compile-html` task calls `browser.reload` after finishing as follows:

```
.on ('finish', browser.reload);
```

Also notice that file changes in the Panini files, except from the `pages` directory. Run the `compile-html:reset` task before the `compile-html` task. The `compile-html:reset` task calls `panini.refresh()` because Panini loads layouts, partials, helpers, and data files once on the first run.

Copying and minifying your images

When your project uses images, you should copy them to the `_site` directory each time you run the build process, because when you run the build process, the clean task removes the temporary `_site` directory.

You can save the images and all other assets in an `asset` directory and run the following task to copy them to the `_site` directory:

```
// Copy assets
gulp.task('copy', function() {
  gulp.src(['assets/**/*']).pipe(gulp.dest('_site'));
});
```

In the case of images, you can consider to not only copy but also minify the images. You can do this with the `gulp-imagemin` plugin.

You can read more about this plugin at https://github.com/sindresorhus/gulp-imagemin

```
var gulp = require('gulp');
var imagemin = require('gulp-imagemin');
var pngquant = require('imagemin-pngquant');
gulp.task('default', () => {
    return gulp.src('assets/images/*')
        .pipe(imagemin({
            progressive: true,
            svgoPlugins: [
                {removeViewBox: false},
                {cleanupIDs: false}
            ],
            use: [pngquant()]
        }))
        .pipe(gulp.dest('_site/images'));
});
```

Putting it all together and creating the default task

At the end of the `Gulpfile.js` file we'll write down some tasks, including the default task, which runs a sequence of tasks of the build process. These tasks may look like the following:

```
gulp.task('set-development', development.task);
gulp.task('set-production', production.task);
gulp.task('test',['scss-lint','validate-html']);
gulp.task('build', ['clean','copy','compile-js','compile-sass','compile-
html']);
gulp.task('default', ['set-development','server', 'watch']);
gulp.task('deploy', ['set-production','server', 'watch']);
```

The default task sets the environment to development by calling the `set-development` task first. Then it runs the `server` task, and starts the watch task after that. Because the `server` task depends on the `build` task, the `build` task always runs before the `server` task (Browsersync) starts. The `deploys` task does the same as the default task, but sets the environment to production in the first call.

The test task lints your SCSS code and checks the compiled HTML code. You can run the test task by running the gulp test command in your console. The result of the test task may look like the following:

```
 [00:15:02] Starting 'scss-lint'...
[00:15:02] Starting 'compile-html'...
[00:15:12] Finished 'compile-html' after 9.25 s
[00:15:12] Starting 'validate-html'...
[00:15:12] Finished 'validate-html' after 22 ms
[00:15:45] 1 issues found in
/home/bass/testdrive/bootstrapcourse/lesson2/scss/includes/_bootstrap.scss
[00:15:45] includes/_bootstrap.scss:1 [W] Comment: Use `//` comments
everywhere
[00:15:45] 2 issues found in
/home/bass/testdrive/bootstrapcourse/lesson2/scss/includes/_page-
header.scss
[00:15:45] includes/_page-header.scss:11 [W] PseudoElement: Begin pseudo
classes with a single colon: `:`
[00:15:45] includes/_page-header.scss:13 [W] TrailingWhitespace: Line
contains trailing whitespace
[00:15:45] Finished 'scss-lint' after 43 s
[00:15:45] Starting 'test'...
[00:15:45] Finished 'test' after 27 µs
[Wed Apr 27 2016 00:15:59 GMT+0200 (CEST)] ERROR
/home/bass/testdrive/bootstrapcourse/lesson2/_site/index.html:58:13 E041
`.carousel` must have exactly one `.carousel-inner` child.
```

Now your build process is ready and you can start using it! To start your build process you'll have to run the following command in your console:

gulp

The `gulp` command runs the default tasks. It starts a static webserver and automatically watches for file changes.

Using the build process to finish your project

At the start of this manual, I have shown you a screenshot of a responsive mobile-first one-page marketing site built with Bootstrap. In the rest of this manual, I will guide you to build this site using the Gulp build process you have created in the preceding sections.

You will have to split up the HTML page into different parts in accordance with the HTML template structure you have already created.

The project will have a single breakpoint at 768 pixels. For viewports wider than 768 pixels, the navigation menu will become horizontal and other adoptions are made.

The layout template

As already mentioned, you'll use Panini to modularize your HTML code. Panini helps you to avoid code duplications in your HTML code.

Panini is powered by the Handlebars templating language. I highly recommend you to read Panini's documentation at the following URL:
http://foundation.zurb.com/sites/docs/panini.html.

The `index.html` file, our main page, will only contain the following content:

```
---
layout: default
title: Home
---
{{> features}}
```

Apart from the `includes/features.html` page, included with the `{{> features}} snippet`, the `index.html` file loads all other HTML code from the default template (`layouts/default.html`). The default template is a common layout that every page in your design shares.

The code of the default template may look like the following:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <!-- Required meta tags always come first -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no">
    <meta http-equiv="x-ua-compatible" content="ie=edge">

    <title> Your Company :: {{title}}</title>

    <!--Bootstrap CSS -->
      <link rel="stylesheet" href="{{root}}css/app.css">
  </head>
  <body>
    {{> header}}
    {{> navigationheader}}
    {{> body}}
    {{> footer}}
```

```
    {{> footerjavascripts}}
  </body>
</html>
```

In this template the `{{> body}}` snippet includes the HTML code of the `index.html` file. The `index.html` file is the first page of your project. The layout default declaration tells Panini to use the default template found in the `html/layouts/default.html` file.

And the `{{> header}}` snippet includes a HTML partial in your page. The partial included with the `{{> header}}` declaration can be found in the `html/includes/header.html` file.

In the next section, you'll develop the page header. Before you start, run the `gulp` command, which runs the default task, and you will directly see the results of your work in your browser.

The page header

Resize your browser so that your viewport is smaller than 768px. The page header should look like that shown in the following screenshot:

Start writing the HTML code for your page header in the `html/includes/header.html` file.

The first version of the HTML code should look like the following:

```
  <header class="page-header">
      <div class="container">
        <div class="row">
        <div class="col-xs-12"><h1 class="display-
4">Your<span>Company</span></h1></div>
        <div class="col-xs-12">
            <form class="form-inline">
            <input class="form-control" type="text" placeholder="Search">
            <button class="btn btn-outline-success"
type="submit">Search</button>
            </form>
        </div>
        </div>
      </div>
  </header>
```

The HTML starts with a header element with a `page-header` class. After the `<header>` element follows a `<div>` element with the `container` class. Containers are the most basic layout element in Bootstrap and are required when using the grid system. The responsive grid system has 12 columns and four breakpoints. The breakpoint defines the five grids: the extra small grid, the small grid, the medium grid, the large grid and the extra large grid. Inside the container, we create a row as a `<div>` element with the `row` class. Columns inside the row are set with `col-{grid}-*` classes.

In the row, we create two columns. The first column contains the company name, and the second column the search form. On small viewports this columns should span 100% of the viewport and be stacked. We'll use the `col-xs-12` class to accomplish that. The naming of the `col-xs-12` class means spanning 12 columns on the extra small (xs) grid. The `col-xs-12` class (although it sets `width:100%`; and `float: left`;) equals the default situation due to the `box-sizing: border-box`

declaration. In the border-box model, block-level elements occupy the entire space of its parent element (container). For this reason, the `col-xs-12` class can be considered as optional.

For the company name a `<h1>` element with a `display-3` class has been used. We'll also use a `<span>` element inside the `<h1>` to enable us to use two colors for the company name.

For the search form in the second column, a default inline form, set with the `form-inline` class and an outlined button, are used.

Now with your HTML code, you can start creating your custom CSS.

Custom CSS code for the page header

Of course, we'll use Sass to build our custom CSS code. For the page header create a new `scss/includes/_page-header.scss` Sass partial. This partial should be imported in the `scss/app.scss` file:

```
@import "includes/page-header";
```

Import the partial after Bootstrap so you can reuse Bootstrap mixins and extend the Bootstrap classes when needed. The SCSS code in the scss/includes/_page-header.scss file may look like the following:

```
.page-header {
  background-color: $page-header-background;

  .display-3 {
    color: $company-primary-color;
    span {
      color: $lightest-color;
    }
  }

  [class^="col-"]:last-child {
    margin-top: $spacer-y * 2;
  }
}
```

Instead of the [class^="col-"]:last-child selector in the preceding code, you can also use a new selector/class, for instance a .search-form-column class, and change your HTML code according to this new CSS class.

The color variables used in the preceding SCSS are declared in the scss/includes/_variables.scss file as follows:

```
$darkest-color: #000; // black;
$lighest-color: #fff; // white;
$company-primary-color: #f80; //orange;

// Page Header
$page-header-background: $darkest-color;
```

Fine tuning of your CSS and HTML code

When you resize your browser smaller than 544 pixels you'll find some issues.

First the search button wraps to the next line. Bootstrap's `inline-form` class only works for screens wider than 544 pixels. You can overwrite the Bootstrap default behavior and use the following SCSS code:

```
.page-header {
  .form-inline .form-control {
    display: inline-block;
    width: auto;
  }
```

```
}
```

Alternatively, you can remove the search button for the smallest screen by adding the `hidden-xs-down` class. The `hidden-xs-down` class makes elements invisible on only the extra small (xs) grid.

Secondly, you'll possibly find that the font size of the company name is too large to fit the smallest screens. You can use the `media-breakpoint-down()` mixins to make the font size smaller for the narrowest screens:

```
.page-header {
  @include media-breakpoint-down(xs) {
    .display-3 {
      padding-bottom: $spacer-y;
      font-size: $display4-size;
      text-align: center;
    }
  }
}
```

Then you should switch to the larger screen, and resize your browser to a screen width wider than 768 pixels.

The columns should become horizontal with each having 50% (six columns) of the available space inside the container. On each grid, the container has a fixed width; for screens wider than 1200 pixels, the container always has a width of 1140 pixels.

You can accomplish this by adding the `col-md-6` class twice:

```
        <div class="col-xs-12 col-md-6"><h1 class="display-
3">Your<span>Company</span></h1></div>
        <div class="col-xs-12 col-md-6">
```

The search box should float on the right side of the header. The right floating is done with the `pull-md-right` class as follows:

```
<form class="form-inline pull-md-right">
```

The `pull-md-right` helper class sets a float: right for the medium grid and up, so the compiled CSS code looks like this:

```
@media (min-width: 768px) {
  .pull-md-right {
    float: right !important;
  }
}
```

While building the page header, you have possibly noticed that the company name and search box may overlap on the medium grid for screen width between 768 and 992 pixels. You can solve this issue by replacing the `col-md-6` classes with the `col-lg-6` classes or alternatively modifying the CSS code so that the font size of the company name is smaller for a larger range of breakpoints. You can use the SCSS code, in the `scss/includes/_page-header.scss` file, as follows, to do this:

```
 @include media-breakpoint-down(md) {
  .display-3 {
    padding-bottom: $spacer-y;
    font-size: $display4-size;
    text-align: center;
  }
 }
```

And finally, you can do a last tweak in the `scss/includes/_page-header.scss` file to apply the `margin-top` for the search box that is set only for the larger screens:

```
@include media-breakpoint-up(md) {
  [class^="col-"]:last-child {
```

```
        margin-top: $spacer-y * 2;
      }
    }
```

Styling the navbar and hero unit

The navbar and the hero unit are wrapped in a `<section>` element, which has the background image. The HTML code in the `html/includes/navigationheader.html` template looks as follows:

```
<section class="nature">

        <header>
        {{> navbar}}
        </header>

        {{> jumbotron}}
</section>
```

Create a new `scss/includes/_navigationheader.scss` Sass partial, which contains the following SCSS code:

// free photo from https://www.pexels.com/photo/landscape-nature-sunset-trees-479/

.nature {

 background-image:url('/images/landscape-nature-sunset-trees.jpg');

}

Notice that you should save a copy of the `landscape-nature-sunset-trees.jpg` photo file in the `assets/images/` directory of your project. As already made clear, this image file is automatically copied to the temporary `_site/images` directory every time you run the build process.

For the navbar, we'll create a `html/includes/navbar.html` HTML template and an `scss/includes/_navbar.scss` Sass partial too. In these files, we reuse the code for the responsive navbar.

You should change the `scss/includes/_navbar.scss` file to set the background color of the navbar:

```
.navbar {
  //background: transparent;
  background: rgba(255,255,255,0.5);
  //@include gradient-horizontal(green, white);
}
```

Notice that in the preceding SCSS code two alternative backgrounds are commented out. You can try the alternative by removing the comment (and commenting out the others). If you have run the `gulp` command already you'll see the effects directly in your browser.

Consider saving the background value in a variable and save it in the `scss/includes/_variables.scss` file for code that can be more easily reused:

$navbar-background: rgba(255,255,255,0.5);

There is also nothing wrong with creating a new class for the navbar's background color:

```
.bg-nature {
 background: $navbar-background;
}
```

Then your HTML code should appear as follows:

<nav class="navbar navbar-dark bg-nature navbar-full" role="navigation">

For the hero unit, we'll use the HTML code for the Jumbotron component directly from Bootstrap's documentation. Save the HTML code into the `html/includes/jumbotron.html` file. Also now create a `scss/includes/_jumbotron.scss` Sass partial.

The HTML code in the `html/includes/jumbotron.html` file will look like the following:

```
        <div class="container">
          <div class="jumbotron">
            <h1 class="display-3">Hello, world!</h1>
            <p class="lead">This is a simple hero unit, a simple jumbotron-
style component for calling extra attention to featured content or
information.</p>
            <hr class="m-y-2">
            <p>It uses utility classes for typography and spacing to space
content out within the larger container.</p>
            <p class="lead">
              <a class="btn btn-primary btn-lg" href="#" role="button">Learn
more</a>
            </p>
          </div>
        </div>
```

Beside the `container`, `display-*` and `btn-*` classes you'll also find some new Bootstrap CSS classes. The `lead` class simply makes a paragraph stand out. The `<hr>` element has a `m-y-2` helper class, which sets the horizontal margins to two times the height of the `$spacer-y` value. The `m-y-2` helper class is an example of Bootstrap's utility classes, which enable you to set the spacing (padding and margin) of an element.

These utility classes are named using the format `{property}-{sides}-{size}`, where property is either p (padding) or m (margin), sides are l (left), r (right), t (top), b (bottom), x (left and right), or y (top and bottom), and the size is a value between 0 and 3, including 0 and 3, meaning {size} time `$spacing-x` or `$spacing-y`.

Then edit the following SCSS code in the `scss/includes/_jumbotron.scss` file:

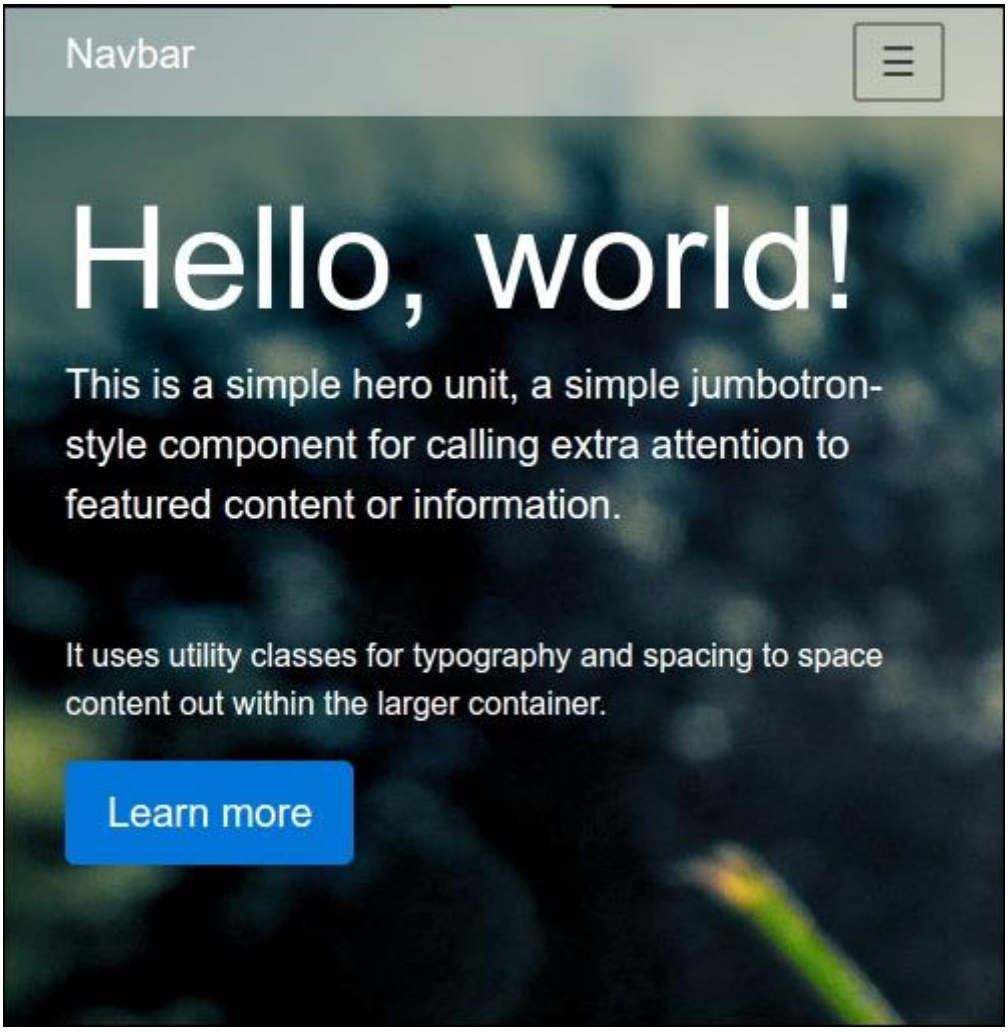
```
.jumbotron {
  background-color: transparent;
  color: $lightest-color;
}
```

On smaller screens the navigation header should look like that shown in the following screenshot:

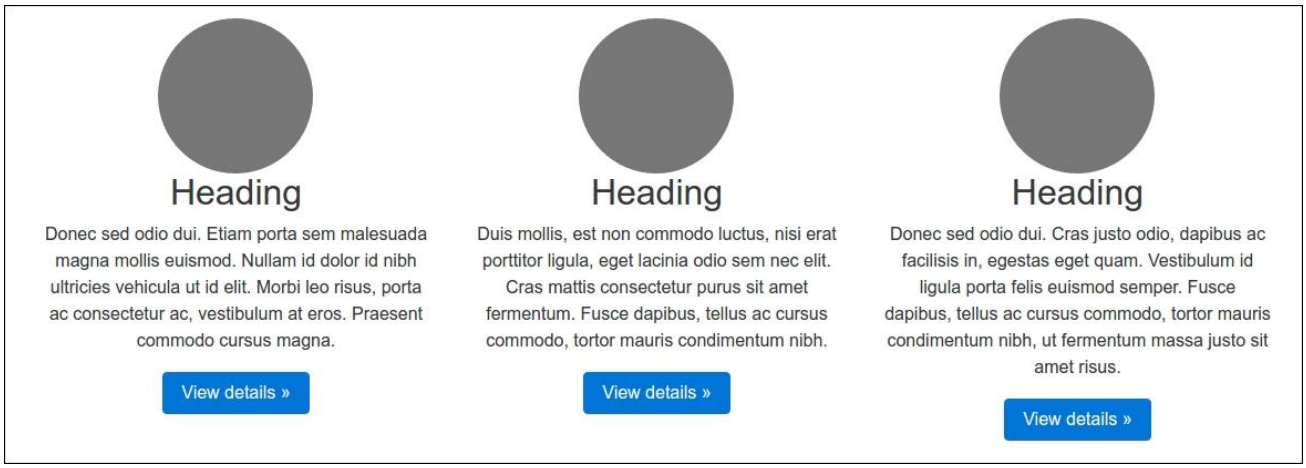
Now the navigation and hero units are ready and we can start styling the features as a part of our design.

Styling the features

Directly under the hero unit, the design shows three features. A single feature may look like that shown in the following screenshot. Each feature starts with a rounded image, which may contain a photo or icon followed by a heading and a text paragraph. At the end, you will find a button which is a **Call to action.**



On screens smaller than 768 pixels, the features should display under each other. On wider screens, the features become horizontal and form three equal width columns like those shown in the following screenshot:



Again we create two files: the `html/includes/features.html` HTML template and a `scss/includes/_features.scss` Sass partial.

The structure of the HTML code in the `html/includes/features.html` HTML template may look like the following (notice that in the HTML code, the features are left out):

```
<div class="container features">
  <div class="row">
    <div class="col-md-4">
      <!-- first feature -->
    </div><!-- /.col-md-4 -->
    <div class="col-md-4">
      <!-- second feature -->
    </div><!-- /.col-md-4 -->
    <div class="col-md-4">
      <!-- third feature -->
    </div><!-- /.col-md-4 -->
  </div><!-- /.row -->
</div>
```

As you can see, we'll use the Bootstrap grid again to display the features. You already know the container and row classes. The `col-md-4 class` creates a space spanning 4 of 12 grid columns on the medium (`md`) grid and up. The optional `col-xs-12` classes are left out here.

The HTML code for each feature may look like the following:

```
            <img class="img-circle"
src="...CRAEAOw==" alt="Generic placeholder
image" height="140" width="140">
            <h2>Heading</h2>
            <p>Donec sed odio dui. .....</p>
            <p><a class="btn btn-primary" href="#" role="button">View details
&raquo;</a></p>
```

The `img-circle` class automatically creates rounded views of your images. You can replace the `src="...CRAEAOw=="` image source with your own images.

> **Note**
>
> Read more about using the data-URIs for your images at https://css-tricks.com/data-uris/.

Now you'll have to create some CSS code to tweak the features' look and feel. Write down the following SCSS code in the `scss/includes/_features.scss` file:

```
.features {
  padding-top: $spacer-y;

  [class^="col-"] {
    text-align: center;
  }
}
```

Instead of creating some new CSS code you can also consider to use Bootstrap's predefined CSS classes to accomplish the same.

You can use the utility classes to set the padding-top of the features as follows:

And the `text-xs-center` class can be used to center the content of the features on all viewports:

Now the features are ready, too. Time to style the footer and finish your project.

Styling the footer of your page

Last but not least, we'll have to style the footer links to finalize our project. Again create an HTML template and a Sass partial to do this.

The `html/includes/footer.html` HTML template should contain HTML like the following:

```
<footer class="page-footer">
      <div class="container">
        <div class="pull-xs-right">
            <a href="https://twitter.com/bassjobsen"><i class="fa fa-
twitter fa-lg"></i></a>
            <a href="https://facecourse.com/bassjobsen"><i class="fa fa-
facecourse fa-lg"></i></a>
            <a href="http://google.com/+bassjobsen"><i class="fa fa-google-
plus fa-lg"></i></a>
            <a href="http://stackoverflow.com/users/1596547/bass-jobsen"><i
class="fa fa-stack-overflow fa-lg"></i></a>
            <a href="https://github.com/bassjobsen"><i class="fa fa-github
fa-lg"></i></a>
        </div>
        <div>&copy; 2016 Bass Jobsen &middot; <a href="#">Privacy</a>
&middot;<a href="#">Terms</a></div>
      </div>
</footer>
```

The footer of our page is simple and straightforward. It contains some copyright notes and a block of social links. The social links are floated on the right side of the footer by the `pull-xs-right` class. The icons of the social links are from the Font Awesome icon font. The `fa-*` CSS classes are not part of Bootstrap's CSS code.

In *Lesson 8*, **Bootstrappin' Your Portfolio**, you can read about how to compile Font Awesome's CSS code into your local CSS by using Sass. Simply load Font Awesome's CSS code from CDN by linking it in the `html/layouts/default.html` HTML template as follows:

```
      <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/font-
awesome/4.6.1/css/font-awesome.min.css">
```

All that remains to do is to give the background, links, and icons for the right colors and add some spacing. You can do this by entering the following SCSS code into the `scss/includes/_ page-footer.scss` Sass partial:

```
.page-footer {
  background-color: $page-footer-background;
  color: $lightest-color;
  a {
    @include plain-hover-focus {
      color: $lightest-color;
    }
  }
  padding: $spacer-y 0;
  margin-top: $spacer-y;
}
```

The preceding code uses Bootstrap's `plain-hover-focus()` mixin which sets the plain, hovered, and focused states at once and uses the Media Queries Level 4 hover `@media` feature as discussed previously.

Again notice that the margin and padding can also be set by using Bootstrap's utility classes. Using `p-y-1` and `m-t-1` will have the same effect:

```
<footer class="page-footer p-y-1 m-t-1">
```

That's all! You did a great job till now!

Running your template with Bootstrap CLI

Here, you have created a build process which builds and compiles your single page template by running the `gulp` command. In this course, you'll use Bootstrap CLI to set up your projects.

Bootstrap CLI enables you to choose your own starter templates. These starter templates also have a Gulp (or Grunt) build chain. For instance the source code of the Bootstrap material design template can be found at https://github.com/bassjobsen/bootstrap-material-design-styleguide. Bootstrap CLI downloads the templates from GitHub and its commands call `npm` to run scripts.

To make your build process and template ready for Bootstrap CLI, you simply add the `npm` script calls to your `package.json` file as follows:

```
"scripts": {
  "build": "gulp deploy",
  "start": "gulp"
}
```

JavaScript task runners are not required

Here, you have learned how to create a build process with Gulp. Both Gulp and Grunt are JavaScript task runners for Node.js. In the previous section you could see that Bootstrap CLI calls npm to run your scripts. For the same reason, some people claim that you can create a build process without Gulp or Grunt too.

> **Note**
>
> Read more about npm script objects at http://blog.keithcirkel.co.uk/why-we-should-stop-using-grunt/.

Publishing your work on GitHub

Since your template is ready to use, you may consider publishing it on GitHub. Other people may use your work, but on the other hand, they can also help you to improve it.

> **Note**
>
> Read more about publishing your project on GitHub at https://guides.github.com/introduction/getting-your-project-on-github/.

Since you have installed both the Bower packages and the Gulp plugins with the `--save-dev` option, your `bower.json` and `package.json` files contain an up-to-date list with the project dependencies. When publishing your project, you do not have to publish the dependencies too. People can download your project files and then install the dependencies by running the following commands:

```
bower install
npm install
```

After the running the `install` command, they can run your project with the `gulp` command; also the `bootstrap watch` command of Bootstrap CLI should work.

You can create a `.gitignore` file to ensure that only your project files are uploaded to GitHub. Your `.gitignore` file should contain the following lines of paths:

```
.DS_Store
bower_components
node_modules
npm-debug.log
```

```
_site
.sass-cache
```