Community Experience Distilled

Getting Started with Zurb Foundation 4

Design and build professional websites with Zurb Foundation's mobile-first responsive framework

Andrew D. Patterson

[PACKT] PUBLISHING open source* community experience distilled

Getting Started with Zurb Foundation 4

Design and build professional websites with Zurb Foundation's mobile-first responsive framework

Andrew D. Patterson

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Getting Started with Zurb Foundation 4

First published: November 2013

Production Reference: 1181113

Published by Packt Publishing Ltd.

Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-596-5

www.packtpub.com

Cover Image by Abhishek Pandey (abhishek.pandey1210@gmail.com)

Credits

Author
Andrew D. Patterson

Reviewers
Vinay Kumar Atthelli
Bass Jobsen

Acquisition Editors
Akram Hussain
Andrew Duckworth

Commissioning Editor
Priyanka S.

Technical Editors
Pooja Arondekar
Rahul U. Nair

Copy Editors
Gladson Monteiro
Kirti Pai
Alfida Paiva

Project Coordinator
Suraj Bist

Proofreader
Maria Gould

Indexer
Tejal Soni

Production Coordinators
Aditi Gajjar
Adonia Jones

Cover Work
Aditi Gajjar

About the Author

Andrew D. Patterson offers services through his own firm Patterson Research Inc., where he provides IT consulting, research, and writing services. He can be available on site in Atlantic Canada and throughout the world, anywhere the Internet reaches.

One of his projects was to help the Internet reach the South Pole.

His entire career has involved technology, from his early days of servicing cash registers to software engineering and consulting. He has worked as a requirements architect, designer, or builder on numerous business and government applications. He understands business, software, and technology. He has worked on both large and small projects. Clients appreciate his independent and thorough perspective when delivering advice and solutions.

For the past 10 years, all of his projects have been Internet-based—from the design and building of websites to open source integrations, medical billing requirements, and business intelligence.

In addition to early electronics training, Andrew has earned a Bachelor of Science degree from the University of New Brunswick, Canada; and he has a Master of Mathematics degree from the University of Waterloo, Canada.

Although he has authored dozens of documents, from requirements ("what is to be") to specifications ("how it should be") and user help ("what is"), this is his first published book.

I'd particularly like to thank my wife, Donna, for her encouragement and support during the writing process. She allowed me a quiet place to work. "What did you get done today?" she would ask. It didn't seem to matter whether she understood the response; it gave me an opportunity to verbalize my progress and to talk about what was next. Thank you, dear.

About the Reviewers

Vinay Kumar Atthelli has more than eight years of experience as a developer of frontend GUIs for web and mobile related applications. He is an avid WordPress developer and a creative person. He graduated from the Osmania University, Hyderabad. He also possess entrepreneurial skills and a keen eye for detail in every project he undertakes.

With extensive knowledge in HTML5, CSS3, and jQuery, he is also enthusiastic about SEO and social media. His interests extend beyond web and he loves helping people. Teaching is his passion—he is interested in teaching technologies and marketing strategies. He also loves building startups and SaaS.

In the words of Wilfried Rijsemus, Director Professional Development and Community at Cordys:

> *"Vinay is too modest if you measure his performance against his demeanor. Do not underestimate the resilience and tenacity in which he drives his products. His work was deemed by a Gartner analyst as the "benchmark to beat" and that quote came unsolicited! He is a hard working team mate who never leaves your side and will always deliver when you need it. He is an excellent wingman, a black hole in learning, and on his way to a mature thought and execution leader. I'm proud to have him on my team."*

Bass Jobsen is from the Netherlands and has been programming on the Web since 1995. From C to PHP, he is always looking for the most accessible interfaces. He has a special interest in the process that takes place between the designing and programming stages. Interfaces should work independent of the device or browser! For these reasons, working with grids and meta languages in designs make him happy. He is looking forward to new opportunities in the semantic and responsive web.

He is always happy to help you at `http://stackoverflow.com/users/1596547/bass-jobsen`.

At the moment he is involved in writing a blog (`http://bassjobsen.weblogs.fm`), programming LBS for mobile devices (`http://www.gizzing.nl`), making cool websites like `http://www.streetart.nl/`, and counseling Jamedo websites (`http://www.jamedowebsites.nl`) on setting up the technical environment and the requirements for their business.

Also, checkout his Bootstrap WordPress Starter Theme and other projects at GitHub (`https://github.com/bassjobsen`).

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

It was only a few years ago that HTML tables were being used heavily for web page layout. Tables have rows and columns; they can stretch vertically and horizontally; rows and columns can be combined into larger spaces. In short, HTML tables made it possible to lay out content on web pages using an underlying grid.

Then along came CSS. Tables were confining, bloated, and did not separate content from presentation, the CSS advocates argued. Style sheets were separate from content. They freed the designer to place content anywhere they wanted on a web page. See the Zen Garden website, `http://www.csszengarden.com`, for a demonstration of how CSS can be used to present the same content in different ways.

Proponents of tables argued that tables were simpler and more predictable than CSS. If you wanted structure on your website, and if you appreciated an underlying grid, it was hard to give up tables. And for all the advantages of using CSS to place content on a web page, it was often used to fix content in place, assuming that all websites would be displayed on a typical computer display (laptop or desktop).

Take a look at the Zen Garden website again, this time from your mobile phone or a very narrow browser view. Most of the views are fixed. Those views were not designed for small devices. Either a page is so compressed that you cannot ready it or you have to scroll horizontally and vertically to see a page.

How do web designers address this dilemma? The natural inclination is to design and maintain two websites, one for regular displays and one for small devices. Many organizations have done exactly that. The .mobi top-level domain was created to give companies an opportunity to make it clear that they were delivering content specifically for mobile devices.

Today there are toolkits that make it easier to design websites using an underlying grid structure using CSS and make those sites adaptable to mobile, desktop, and other displays. Now we have the benefits of a grid layout with all the capability of CSS, the best of both worlds. Zurb Foundation is one—some would argue the best—of those toolkits.

Welcome to Foundation, an amazing toolkit that helps you architect, design, and develop your website. You get a grid system, numerous CSS components and JavaScript plugins, and you have style sheets coded in SASS to ease design and customization.

What this book covers

Chapter 1, *Get the Most from the Grid System*, explains the Foundation grid system, introduces various grid features included in Foundation, shows by example how to apply layout techniques, and offers tips to deal with special cases.

We'll discover how to work with cells, rows, and columns. We'll look at nesting rows in columns. We'll learn about gutters, page width, and row height. We'll see how to apply the grid to small and large devices, use built-in styles to tweak your layout, and also learn what to expect when you put images in your grid.

Throughout the chapter we'll include a few tips and tricks that show how, with just a few styles in a custom style sheet, you can have a nice effect on your design.

Chapter 2, *Pick and Choose CSS Components*, explains the components that only require style sheets.

Foundation components are bits of Cascading Style Sheets (CSS) code that help you style and present website content. Rather than starting from scratch and styling everything yourself, you can start with Foundation's components. Often these will be sufficient as they are; however, sometimes you will want to override or add to the built-in styles to customize a component for your needs.

The only JavaScript you need is the `modernizr.js` package at the top of your page.

We'll cover the following components: typography, visibility, miscellaneous components (keystrokes, inline lists, tables, labels, and vCards), navigation components (breadcrumbs, pagination, side nav, and sub nav), buttons, block grids, thumbnails, panels, flex video, pricing tables, and progress bars. Enjoy!

Chapter 3, *Pick and Choose JavaScript Plugins*, explains that there are several other components available that also require JavaScript—software that runs in the browser. JavaScript plugins can do a lot to enhance the capability of your website, but with all the wonderful features you also get more complexity.

Troubleshooting HTML markup and CSS is one problem. Digging into problems with JavaScript is much more of a challenge. Having said this, Foundation's plugins will work and do amazing things for you, provided you set them up as the plugin expects. This chapter will help you with that.

The plugins we'll cover in this chapter are Alerts, Tooltips, Dropdown, Sections, Top bar, Magellan, Joyride, Orbit, Clearing, Reveal, Interchange, Forms, and Abide.

Chapter 4, *Advance with SASS*, helps you advance with Foundation's SASS. After a brief look at what SASS is about, we'll see how to install Foundation with SASS.

When you download Foundation from the website, you can selectively choose which components to include. We'll show you how to do that after you have installed Foundation with SASS.

Then we'll see how to customize your CSS with SASS variables, how to work with SASS functions and mixins, and how to extend classes using SASS. Each of these allows you greater flexibility and control of your style sheets.

In the last section of this chapter we'll introduce you to designing with SASS. That's where you can take your HTML markup and quickly apply Foundation's grid mixins to style it into the layout you desire.

What you need for this book

Whether you want to use all of Foundation's components or just a few, you can download and install them from the Zurb website. Here is the simplest case:

1. In your browser go to the Foundation download page. It is `http://foundation.zurb.com/download.php` at the time of this writing.
2. Click on the download button. This will download all of Foundation with the default settings to your PC.
3. Unzip the downloaded file and copy the resulting foundation folder to the location where you want it installed.
4. Verify the download by opening the `index.html` file that you find in the root of the foundation folder. You should see **Welcome to Foundation** as a bold heading, **This is version x.x.x.** in regular font just below that, and much more.

By default, the grid system along with all Foundation components and plugins are installed. Use the `index.html` file that you just loaded from the foundation folder as a model for your own pages.

If you already have a website where you want to install Foundation, you can copy the subfolders in your Foundation download into the root folder of your website. That includes the `css`, `img`, and `js` folders. With the current version of Foundation, the `img` folder is empty so it doesn't matter. If your website uses the `css` and `js` folders already, that's good. You can just merge Foundation's files with yours. Just be sure there are no files with the same name so you don't overwrite one with another.

This can be a bit confusing when getting started, and later on it can be hard to remember which files came from Foundation and which came from another source. So you may want to copy the entire foundation folder into the root directory of your website and keep it separate. When you do that, remember to change the links to CSS and JavaScript files.

For example, let's assume you copied the entire Foundation folder into your root directory and named it `foundation`. Copy the `index.html` file from the foundation folder to your root folder and call it `model.html`. Edit that in your favorite text editor. Replace all occurrences of `js/` with `foundation/js/` and all occurrences of `css/` with `foundation/css/`. Open `model.html` in your browser. If everything went well, you will see exactly the same thing you had seen on opening `index.html` from the `foundation` folder.

Now use `model.html` as the model for loading your CSS and JavaScript files.

One thing is critical for the maintenance of your website, never edit the Foundation files you downloaded from the website. Always keep them intact so you can install a new version of Foundation without wiping out your changes.

When you want to override or customize something in Foundation, do it in a separate folder, or at least a separate file. In this book we'll refer to your custom style sheet. It is recommended that you create a custom style sheet (for example, `custom.css`) where you add any tweaks or overrides that you want to make to Foundation's styles.

Put the custom style sheet wherever it makes sense for your website. That could be in Foundation's `css` folder or your own folder. Then for every web page you create, load the custom style sheet immediately after loading Foundation's style sheet.

On Foundation's download page you may have noticed a couple of other download buttons. Ignore the third one for now. We'll cover SASS installation in a later chapter. But the second button, **Customize Foundation,** gives you the option of setting some variables and selecting which components and plugins to install.

When working through the examples in this book it is recommended that you install all components and plugins. But as you get ready to move a website into production, you can optimize it by loading only the CSS and JavaScript that you use. The simplest way to do that is to do a custom download.

When you set variables or selectively install components and plugins, remember your choices. You'll want to go back for an update at some later date and use the same values.

To do an update of Foundation, download a newer version—using the same values if it is a custom download—unzip and copy the new folders over the existing ones, replacing the current installation. It is best to copy the entire website into a test area first, test the update there, and verify that nothing has changed that will break your website. Only when you are satisfied that nothing breaks, or you have fixes in place, should you replace the production Foundation files with a newer version. That's just good practice.

To get the best results with Foundation, include the following style sheets in this sequence in every web page:

- `normalize.css`
- `foundation.css`
- `custom.css`

Foundation ships with the `normalize.css` style sheet, which is part of the HTML5 Boilerplate package. We will not cover the specifics of `normalize.css`. There is documentation in the file itself and on the HTML5 Boilerplate website. Suffice to say that it normalizes styles for many HTML elements, corrects cross-browser inconsistences, fixes bugs, and generally improves usability.

The file `foundation.css` contains Foundation's styles and `custom.css` will contain your customizations.

Who this book is for

The book will be useful to web architects, designers, and builders. While it helps to be a programmer, it isn't necessary. The reader should be familiar with the basic principles of responsive web design and have a desire to create a professional website that looks great on mobile devices and regular displays. A mind that's open to discovering the techniques and components that are available, and a will to put them to work, is all that's required.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Use `small-n` and `large-n` when you want to control the view on small and large devices independently."

A block of code is set as follows:

```
<div class="row">
  <div class="small-10 small-centered column">
    <!-- centered -->
  </div>
</div>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are highlighted:

```
<div class="row">
  <div class="small-10 small-centered column">
    <!-- centered -->
  </div>
</div>
```

Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "The third installation option on that download page is **Install Foundation SCSS.**"

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Get the Most from the Grid System

This chapter explains the Foundation grid system, introduces various grid features included in Foundation, shows by example how to apply layout techniques, and offers tips to deal with special cases.

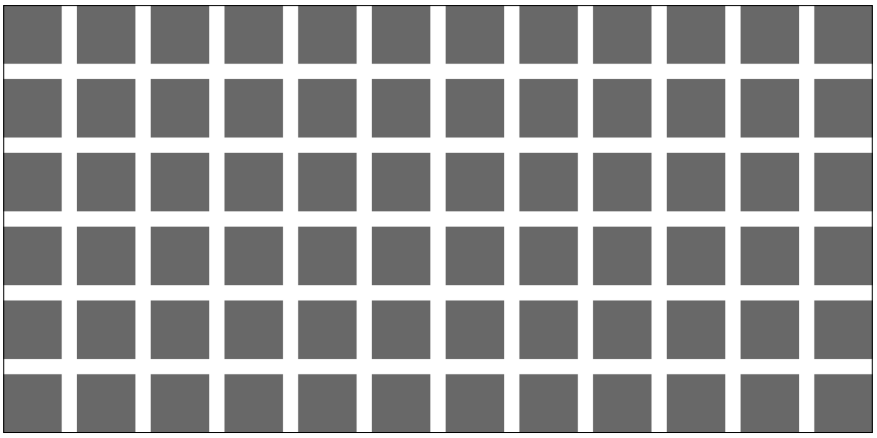
We'll discover how to work with cells, rows, and columns. We'll look at nesting rows in columns. We'll learn about gutters, page width, and row height and see how to apply the grid to small and large devices. We'll use built-in styles to tweak your layout and we'll learn what to expect when you put images in your grid.

Throughout the chapter, we'll include a few tips and tricks that show how, with just a few styles in a custom style sheet, you can have a nice effect on your design.

Working with cells, rows, and columns

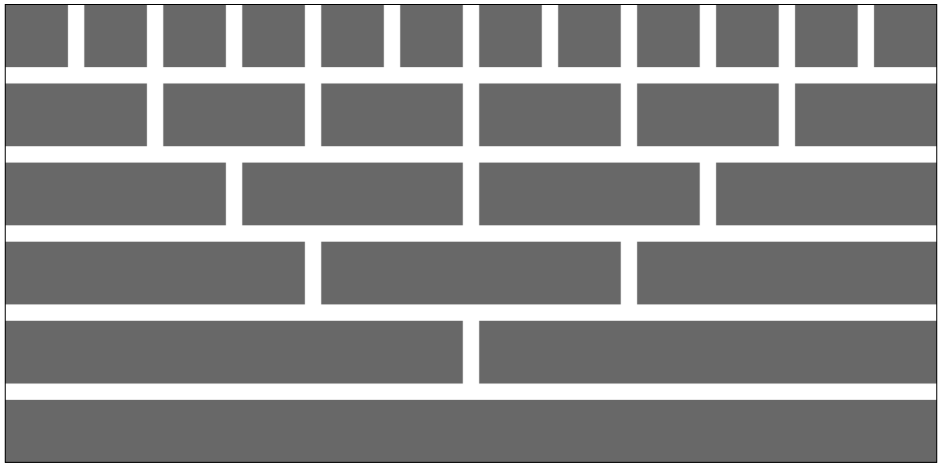
The foundation (pun intended) of Zurb Foundation is its grid system—rows and columns, much like a spread sheet, a blank sheet of graph paper, or tables similar to what we used to use for HTML layout. Think of it as the canvas upon which you design your website.

The following is a way to picture a blank grid before you begin your design:



Each cell is a content area that can be merged with other cells beside or below it to make larger content areas. A default installation of Foundation will be based on 12 cells in a row. This is convenient because the number 12 is divisible by 1, 2, 3, 4, 6, and 12.

The following is the grid again with the cells in a row grouped into 12, 6, 4, 3, 2, and 1 columns respectively:



A column is comprised of one or more individual cells in a row. The total width of the columns in a row cannot exceed the number of underlying cells in the grid's row.

The following figure is an example of a typical blog layout:

How do we create these areas for our website? We simply define rows and columns using HTML. The following is the HTML code for the header row in the example blog:

```
<div class="row">
  <div class="large-6 column">
    <!-- Header left -->
  </div>
  <div class="large-6 column">
    <!-- Header right -->
  </div>
</div>
```

The outer `<div>` tells the browser to form a block representing a row. It gives that block all the styles associated with a row class. Each inner `<div>` tells the browser to form a block representing a column within the row where each column has a width of 6 cells. As our grid has 12 cells in a row, each column with six cells is 50 percent of the row width.

Foundation styles look after keeping the two columns in one row and making them equal in width. We'll cover the `small-n` and `large-n` classes in more detail further on. In this example, specifying each column as `large-6` means that it will use six cells on a regular display.

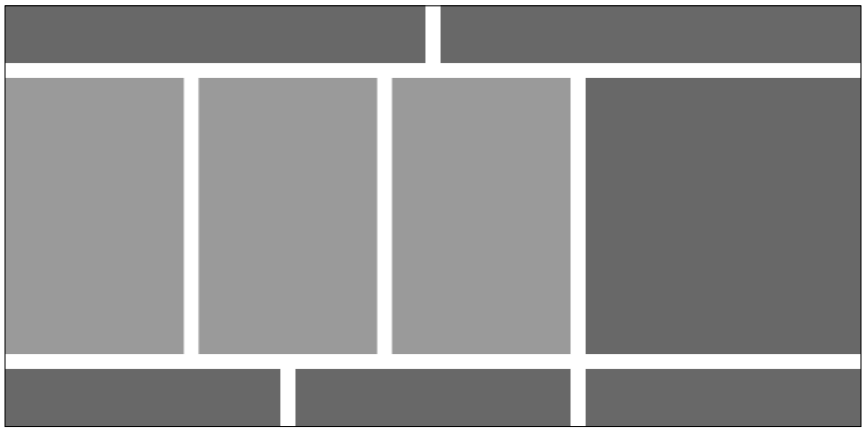
From this, you can write the HTML for the content and footer rows in our sample blog. Each will be a row. The content area has two columns, of which one is eight cells wide and the other is four cells wide. The footer area has three columns, each being four cells wide.

Nesting rows in columns

The real strength of Foundation as a tool for website layout is its nesting capability. Besides a simple grid with rows and columns, you can nest rows within columns. Each newly defined row has its own cells to be grouped into columns.

Going back to our blog, let's say you have something in your content area that would best be displayed in three equal columns. That area was eight cells wide. 8 is not divisible by 3. But 12 is divisible by 3.

So let's create a new row within the content area. We'll refer to this as an inner row as it is treated a little differently than the outer rows we've seen up until now. This is what we want it to look like where the three columns shown in a lighter grey are columns in an inner row:



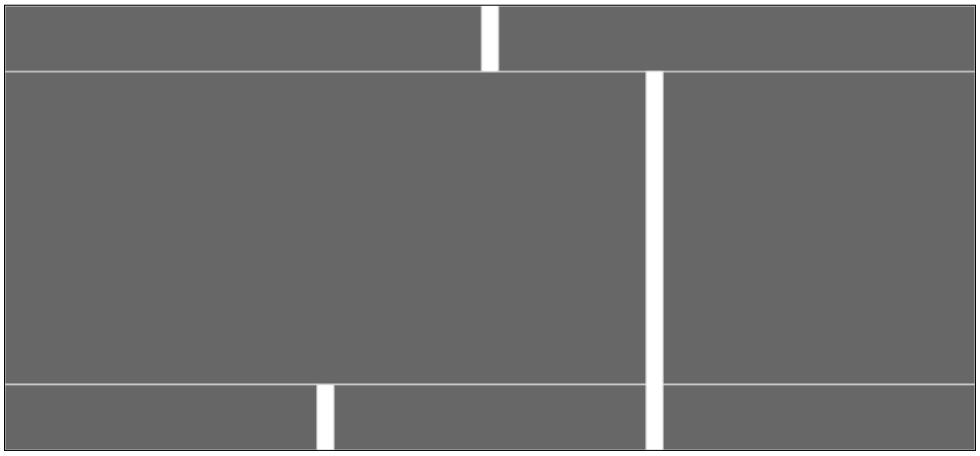
The HTML for the eight-column content area is as follows:

```
<div class="large-8 column">
  <div class="row">
    <div class="large-4 column"><!--content left--></div>
    <div class="large-4 column"><!--content center--></div>
    <div class="large-4 column"><!--content right--></div>
  </div>
</div>
```

Imagine the possibilities. Any space you have on your website can contain a grid with its own rows and columns.

Understanding gutters

By design, Foundation leaves spaces between columns. This space is called a gutter. Foundation does not leave gutter space between rows. That is left for the content to manage. So if we look at the actual space defined by the row and column styles in our original blog layout, it looks like the following figure:



Most content will have margins or padding above or below it. For example, the paragraph tag `<p>` is defined in Foundation as having a margin-bottom property of 1.25 em. So if you are using paragraph tags, you will have a 1.25 em space between rows. But that is derived from the content. It is not inherent in Foundation's grid definitions.

Now what if we want to create a gutter between rows? For example, what if we are putting images inside columns? Images have no inherent space above or below them, so there would be no space between rows. The following custom styles will add a 1 em gutter between rows, half at the top and half at the bottom:

```
.row {
  padding-top: 0.5em;
  padding-bottom: 0.5em;
}
.row .row {
  margin-top: -0.5em;
  margin-bottom: -0.5em;
}
```

To avoid double gutters for nested rows, we added a negative margin at the top and bottom of the nested rows.

Foundation leaves spaces between columns through the use of padding. The following are the values from a default installation:

```
.column, columns {
  padding-left: 0.9375em;
  padding-right: 0.9375em;
}
```

For outer rows, there will be a 0.9375 em space on the left and right. Between the columns, there will be a 1.875 em (2 * 0.9375 em) gutter.

For nested, inner rows, the padding on the left and right columns is omitted. Only the padding between the columns remains. This allows the content of the inner rows to line up within its space. Foundation accomplishes this bit of magic by defining negative margins on inner rows:

```
.row .row {
  margin-left: -0.9375em;
  margin-right: -0.9375em;
}
```

These negative margins on the row are equal to the left and right padding of columns, so they effectively cancel each other out.

Let's say you want to set the gutter to 16 px (1 em) instead of the default 30 px (1.875 em). In your custom style sheet, just set the padding on columns and the margin on inner rows as follows:

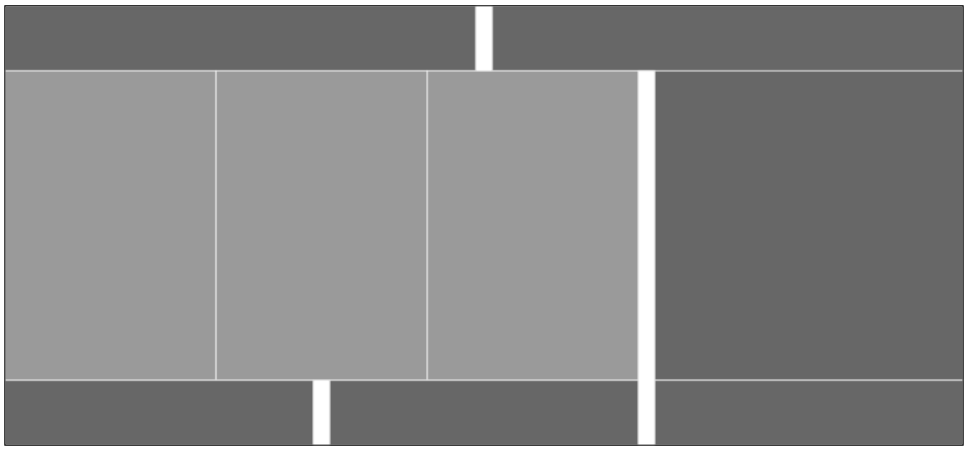
```
.column, .columns {
  padding-left: 0.5em;
  padding-right: 0.5em;
}
.row .row {
  margin-left: -0.5em;
  margin-right: -0.5em;
}
```

There may be times when you want to get rid of the gutters between columns. Foundation makes that easy with the `collapse` class.

For example, to eliminate the gutters from the inner row of our eight-columned content area, simply add the `collapse` class while defining the inner row.

```
<div class="large-8 column">
  <div class="row collapse">
    <div class="large-4 column"><!--content left--></div>
    <div class="large-4 column"><!--content center--></div>
    <div class="large-4 column"><!--content right--></div>
  </div>
</div>
```

We've only included the preceding code for the inner row. Because of the `collapse` class there are no gutters between the columns in the inner row. This is what it will look like:



There are a couple of limitations in having fixed-width gutters. As content is scaled to a smaller display, only the content area in a column scales. The gutters remain fixed. You might prefer for them to scale proportionally along with the column content.

This is particularly noticeable when you have images in columns. As soon as your page is squeezed to less than the full width, each column gets squeezed proportionally. But the columns include padding, which has a fixed width. Narrower images get squeezed a greater percentage than wider ones, and as the height is proportional, it decreases more, thereby putting things out of perspective.

The solution is surprisingly simple. Instead of specifying column padding in em (or px), specify it in percentage. In the following example, gutters are set to 5 percent, half on the left and half on the right:

```
.column, .columns {
  padding-left: 2.5%;
  padding-right: 2.5%;
}
.row .row {
  margin-left: -2.5%;
  margin-right: -2.5%;
}
```

Now, content such as images in columns will scale proportionally. Be sure to include the offset for inner rows so they continue to align properly.

Maintaining row and page width

The simplest way to set the maximum page width for your website is to put all your content inside a containing row.

```
<body>
<div id="container" class="row"><div class="large-12 column">
<!-- website content -->
</div></div><!-- end #container -->
</body>
```

This code creates an outside row with a maximum width equal to the `max-width` value of the row (the default value is 62.5 em or 1000 px).

As a general rule, while creating an outer containing row, always include a column that has the full width (for example, `large-12 columns`). Otherwise, the negative margins of an inner row will make your website wider than you anticipated.

Only the outermost row has a specific maximum width value. Inner rows expand to fill the container in which they reside.

While having a single containing row is the simplest way to set the maximum width for your site, you can also build your site in sections. In the blog we created earlier, we had an outside row for each of the three sections: header, main, and footer.

This also works to keep your site at a fixed maximum width. As long as all the content is within those rows, the content will align itself, and the site width will be limited to the maximum width for the outer `row` class in Foundation.

To override the maximum width of your site in a custom style sheet, simply add a style as follows:

```
.row { max-width: 128em; }
```

That's it. Now your outer rows, and therefore your site width, will use as much as 128 em (2048 px). Choose your own maximum width, larger or smaller than the default 62.5 em.

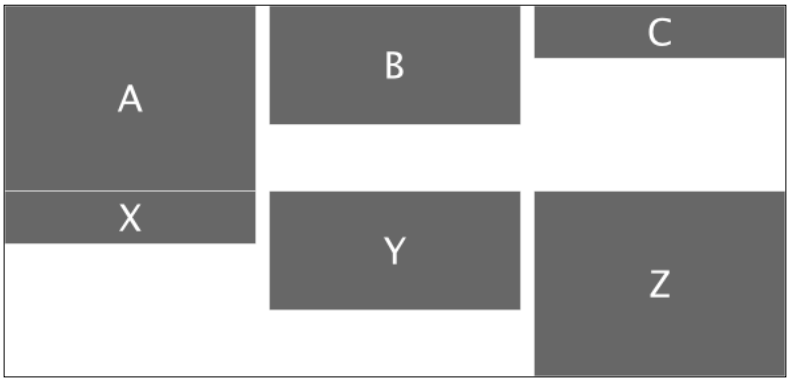
Understanding row and column height

Column height is determined by its content. Row height is determined by the highest column in a row.

The following code snippet shows two rows, each with three columns of different heights:

```
<div class="row">
  <div class="large-4 column"><!-- content A --></div>
  <div class="large-4 column"><!-- content B --></div>
  <div class="large-4 column"><!-- content C --></div>
</div>
<div class="row">
  <div class="large-4 column"><!-- content X --></div>
  <div class="large-4 column"><!-- content Y --></div>
  <div class="large-4 column"><!-- content Z --></div>
</div>
```

The following figure shows the output of the preceding code:

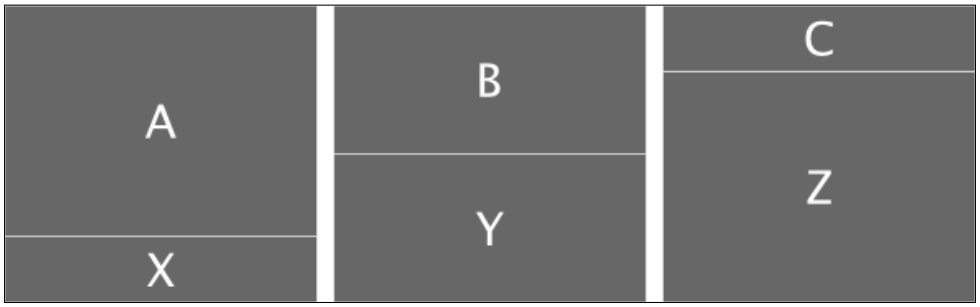


Foundation does not squeeze a column from one row up into the space of another row even when it might fit. Sometimes you are working with content where you just don't know in advance how high a column will be. In those situations, let the content guide you on how it should be organized.

At other times, you may be designing for a more predictable height and decide you want to mesh the columns together. You can display the same content as the preceding example meshed together as follows:

```
<div class="row">
  <div class="large-4 column">
    <!-- content A -->
    <!-- content X -->
  </div>
  <div class="large-4 column">
    <!-- content B -->
    <!-- content Y -->
  </div>
  <div class="large-4 column">
    <!-- content C -->
    <!-- content Z -->
  </div>
</div>
```

The output is shown in the following figure:



Designing small to large

One of the key principles of Zurb Foundation 4 is that you are encouraged to design for small, generally mobile devices first. Then adapt your design for large displays, usually desktops and laptops. This is called mobile-first design.

Intuitively, it just makes sense. The traditional approach was to design for regular displays and make exceptions for larger and smaller devices. It is easier to design for small devices (one end of the spectrum) and make exceptions for larger devices.

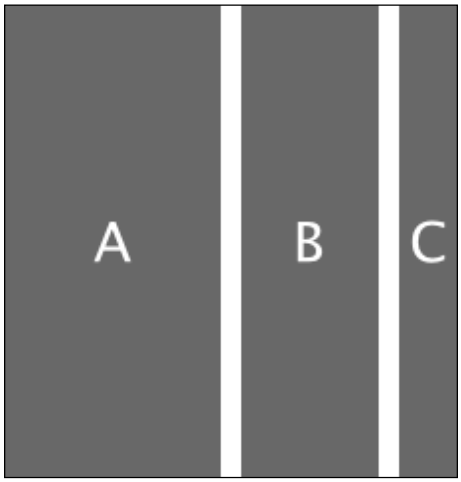
While Foundation is capable of handling multiple breakpoints, in this book we'll concentrate on two sizes, small and large. A breakpoint is a specific browser width that delineates small and large displays. Foundation's default breakpoint is 768 px. When a browser is narrower than the breakpoint, we call it a small display. When it is wider, we call it a large display. The principles are the same when you introduce additional sizes, so if you can manage two, you can manage three.

When we defined columns in our previous examples, we prefixed them with the `large-n` class. We can use either the `small-n` or `large-n` classes. The following are the rules:

- Use `small-n` alone to define the number of cells to use in both small and large devices.
- Use `large-n` alone to define the number of cells to use only on large devices. When the content is then displayed on small devices, it assumes `small-12` and uses all available width.
- Use `small-n` and `large-n` when you want to control the view on small and large devices independently.

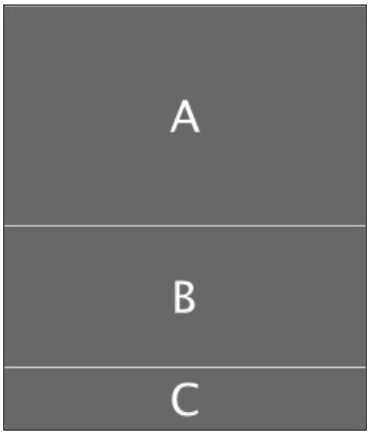
Let's look at some examples. Assume a row with three columns. The first column is six cells wide, the second is four cells wide, and the third is two cells wide.

If we specify all three columns with the `small-n` classes, then on a small device, the content will appear narrow and require more vertical height as shown in the following figure:



When viewed on a desktop or laptop, the relative width of each column is the same, but the content will spread out horizontally and require less vertical height as shown in the following figure:

If we specify all three columns with the `large-n` classes, then on a small device each column takes the full width of the screen because nothing was specified for `small-n`, and it defaults to full width, as follows:



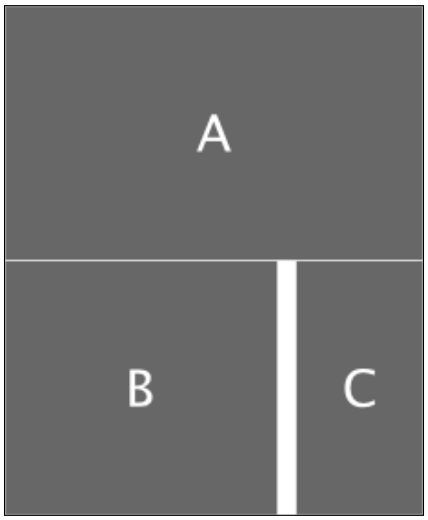
When viewed on a desktop or laptop, it will look exactly as it did when only the `small-n` columns were defined.

We can specify both the `small-n` and `large-n` classes separately as follows:

```
<div class="row">
  <div class="small-12 large-6 column"><!-- content A --></div>
  <div class="small-8 large-4 column"><!-- content B --></div>
  <div class="small-4 large-2 column"><!-- content C --></div>
</div>
```

When viewed on a small device, column A takes the full width of the screen because it was specified as `small-12`. Columns B and C flow under column A. Column B is `small-8` and column C is `small-4`, so between them they take up the full width. We have columns totaling 24 cells in the small view. That's ok. Foundation looks after wrapping. As soon as a column exceeds the grid maximum (12 here), subsequent columns wrap into a new row.

The preceding code's output will look as follows on small devices:



When viewed on a desktop or laptop, it will look exactly as it did when only the `small-n` or `large-n` columns were defined.

How does Foundation know whether it is a small or large device that is being used to display content? It uses media queries.

A media query is a conditional statement used in a style sheet in such a way that styles within the media query take effect only if that condition is true. For small devices, styles are defined outside of a media query. This way they remain in effect unless there is an override within a media query. Therefore, all styles specific to large devices must be within a media query.

The following is the media query for large devices:

```
@media only screen and (min-width: 768px) {
// .large-n styles
}
```

Any devices that report a browser width of `768px` or greater will use the `large-n` classes when they are used in the markup. Otherwise, they will default to the classes defined outside the media query.

Can we always rely on media queries? Unfortunately, the answer is no. Newer browsers and the latest devices all support media queries. So for the majority of Internet users, media queries are fine.

But if you are in an organization where you must support older browsers (for example, IE8 and earlier), look for JavaScript add-ins that emulate media queries for those browsers. Or at the very least, do a bang-up design for small devices because that's what users with old browsers that do not support media queries will see.

Tweaking the layout

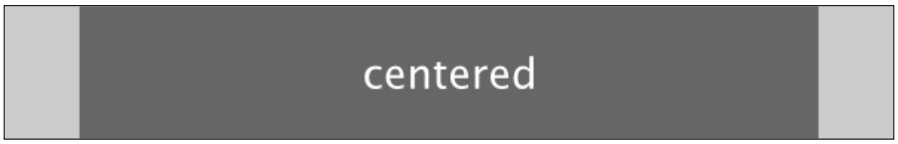
There are a few additional things you can do to adjust your layout within the grid. These include centering, offsetting, and changing the order of columns.

Centering columns

There may be times when you want to center some columns within your grid. For example, by creating a column with fewer than twelve cells and centering that column, you effectively create a margin on either side. The following is an example:

```
<div class="row">
  <div class="small-10 small-centered column">
  <!-- centered -->
  </div>
</div>
```

This will create a ten-cell column that is centered in the row as follows:



When you use the `small-centered` class alone, it centers the column on small devices and regular displays. If you want your column to float left on small devices and be centered only on regular displays, add the `large-centered` class alone.

If you want your column to float right on small devices and be centered on regular displays, add the `right` and `large-centered` classes.

If you want your column to be centered on small devices and float left on regular displays, add the `large-uncentered` class.

If you want your column to be centered on small devices and float right on regular displays, add the `large-uncentered` and `opposite` classes.

Rearranging columns

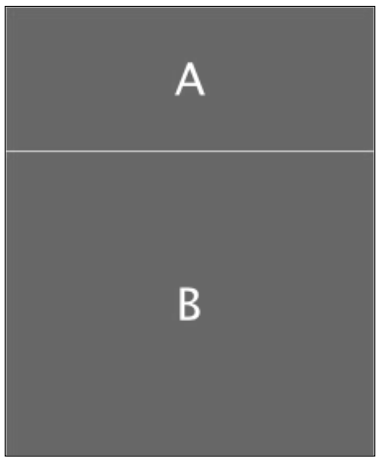
Sometimes, you may want to display your columns in one order for small devices and a different order for regular displays. A simple example of this would be when you want your blog sidebar content to be at the top on small devices, and you want it on the right on regular displays.

Foundation provides a way to do this through the use of the `push-n` and `pull-n` classes.

Browsers will display content in the order it is delivered, unless you override the sequence. So the idea is to deliver sidebar content A, before blog content B. The small device will display it in that sequence:

```
<div class="row">
  <div class="small-12 column">A</div>
  <div class="small-12 column">B</div>
</div>
```

Because we've made each column `small-12`, each one will take up the full width on small devices and display one above the other:



But on a regular display, we want to display the sidebar content A to the right of blog content B. This is the opposite of the order in which it is delivered to the browser as follows:

```
<div class="row">
  <div class="small-12 large-3 push-9 column">A</div>
  <div class="small-12 large-9 pull-3 column">B</div>
</div>
```

The preceding code results in the following layout:

The `push-9` class pushes content A nine cells to the right of where it otherwise would be by default. The `pull-3` class pulls content B three cells to the left of where it otherwise would be.

As an aside, you may recall that the `small-12` class is the default when we have a `large-n` class, so it could have been left out of this example.

This was a simple example. You can push and pull when you have multiple columns and within nested rows. It is a very useful technique for managing the presentation of the same content on mobile devices and regular displays.

The `push-n` and `pull-n` classes only affect the order on regular displays. You can still use the `small-n` class to define columns for both. The following code will display A to the left of B on small devices and B to the left of A on regular displays:

```
<div class="row">
  <div class="small-3 push-9 column">A</div>
  <div class="small-9 pull-3 column">B</div>
</div>
```

A subtle point in understanding source ordering is that the push and pull classes only affect the column where they are specified. They do not affect prior or subsequent columns unless those columns also have a push or pull class. So you typically have at least one pull for every push, and vice versa. Otherwise, you would simply slide one column under another.

Applying offsets

You would use an offset when you want to leave an empty space in your row.

An `offset-n` class is similar to a `push-n` class, in that it moves a column some number of cells to the right. The difference is that it also moves all subsequent columns to the right. An offset is like creating an empty column, the width of the offset.

A simple use of offsets would be to indent the content:

```
<div class="row">
  <div class="small-offset-2 small-10 column">Indented 2</div>
  <div class="small-offset-4 small-8 column">Indented 4</div>
</div>
```

The following figure shows how this would appear:

The first column has a width of 10 cells and is offset 2 cells, so it uses the full width of the row. The second column has a width of 8 cells and is offset 4 cells, so it also uses the full width of the row.

Offsets can also be used anywhere in a row to create space between columns as shown in the following code snippet:

```
<div class="row">
  <div class="small-2 column">A</div>
  <div class="small-offset-2 small-4 column">B</div>
  <div class="small-offset-2 small-2 column">C</div>
</div>
```

This code produces empty space as shown in the following figure:

Unlike push and pull classes, which only apply to regular displays and override the default order of columns on small devices, offsets can be defined for small and large devices separately.

Similar rules apply for the `small-n` and `large-n` columns. When `small-offset-n` is used alone, it has the same effect on small and large displays. When `large-offset-n` is used alone, it is ignored on small displays and only affects large displays. When both `small-offset-n` and `large-offset-n` are used, each of them affects its respective display.

Foundation has a built-in offset that you should be aware of. When you use fewer cells in a row than there are available, the last column will float to the right, effectively leaving an offset equal to the number of unused cells.

In the following figure, three columns are defined. A is four cells wide, B is four cells wide, and C is two cells wide. Column C floats to the right as if it had a `small-offset-2` class defined for it.

To change the default behavior on the last column and shift it back to the right of the column preceding it, you can add the `end` class to the last column:

```
<div class="row">
  <div class="small-4 column">A</div>
  <div class="small-4 column">B</div>
  <div class="small-2 column end">C</div>
</div>
```

The preceding code shifts column C back so that it is next to column B as follows:

Working with images

While designing a website, it is important to understand how images will be presented by default so you can adjust when necessary. In a Foundation-based site, where images are inside a row and column, this is especially true.

An image has an inherent width and height. Styles on the `<img>` tag will determine how an image will be presented in the browser. The following are Foundation's defaults:

```
img {
  max-width: 100%;
  height: auto;
  display: inline-block;
  vertical-align: middle;
}
```

When presented with these defaults, an image will use all the space that it can while being constrained by that space.

The following is an example code that demonstrates these styles:

```
<div class="row">
  <div class="small-3 column">
    <img src="http://placehold.it/220x220">
  </div>
  <div class="small-3 column">
    <img src="http://placehold.it/100x400">
  </div>
  <div class="small-3 column">
    <img src="http://placehold.it/400x400">
  </div>
  <div class="small-3 column">
    <img src="http://placehold.it/100x100">
    <img src="http://placehold.it/100x300">
  </div>
</div>
```

We started with a grid that was 1000 px wide and created four equal columns that were 250 px wide. With 15 px padding on each side, there is 220 px left for content.

The following is a screenshot of the result when shown in a browser that is wider than 1000 px and therefore shows the content at its maximum width:

The image in the first column exactly fits the width of the column, which is 220 px. The column height (medium gray) is exactly the height of the image.

The second column contains an image having width 100 px and height 400 px. As the width of the image is less than the width of the column, the full width of the image is displayed, and therefore, the full height is also displayed. As this is the highest column, it determines the height of the row, which is also 400 px.

The third column contains an image having width 400 px and height 400 px respectively. This is wider than the available 220 px, so the image width is constrained to 220 px. As the height is adjusted proportionally to the width, it is also 220 px. This image is displayed having exactly the same size as the image in column one, even though the underlying image is larger.

The fourth column contains two images. The first one is 100 px by 100 px and the second one is 100 px by 300 px. As each one is only 100 px wide, there is sufficient width in the column to display them side by side. Because the full width of each image is displayed, the full height is also displayed. This makes the column 300 px high, and the height of the taller image and the smaller image sits in the middle vertically. Of all the images shown here, this smaller image is the only one that does not determine its column height.

Summary

If you could learn only one thing from this book, the most important would be to understand the grid system. It is the basis for web page layout with Foundation, and with it, you can create professional sites with ease.

In this chapter we learned how Foundation's grid system is comprised of cells, rows, and columns. We learned to manipulate rows, columns, and the space between them to get different effects. We learned how content in general, images in particular, both influence and are constrained by the grid system. Throughout, we took home some tips that allow us to create our own styles to improve control over our grid-based layout.

Now that you've mastered the grid system, in the next chapter we'll dig into all the additional components included in the Foundation toolkit that only require style sheets to work. You may not use all of them right away, but it helps to be aware of the possibilities, how to put them to work, and how to tweak them to your preferences.

2
Pick and Choose CSS Components

Now that we are familiar with the grid system, it is time to see what else we can do with Zurb Foundation; it is a comprehensive toolkit.

Foundation components are bits of **cascading style sheet** (**CSS**) code that help you style and present website content. Rather than starting from scratch and styling everything yourself, you can start with Foundation's components. Often, you can use these as provided by Foundation. Other times, you will want to override or add to the built-in styles to customize a component for your needs.

The more complex components also require JavaScript. In this chapter, we'll look at the components that only require style sheets. The only JavaScript you need is the `modernizr.js` package at the top of your page. You do not require any of the Foundation-specific scripts that are usually included at the bottom of your web page.

In this chapter, we'll cover the following components: typography, visibility, miscellaneous components (keystrokes, inline lists, tables, labels, and vCards), navigation components (breadcrumbs, pagination, side nav, and sub nav), buttons, block grids, thumbnails, panels, flex video, pricing tables, and progress bars. Enjoy!

Typography

For our purposes here, typography is the art of presenting text and other HTML elements in such a way that they are meaningful and look good to the viewer.

Global styles

Irrespective of whether you have installed one or more components, there are some basic global styles that will always be included. You'll find these at the top of the Foundation CSS style sheet.

These include things such as default styles for the `<html>` and `<body>` tags, default font styles for most HTML elements, and some classes to make images and other objects work well with the grid system.

This section of the style sheet also has a few global classes that are handy when you need them. These include the following:

- `left`: This makes a container float left
- `right`: This makes a container float right
- `hide`: This hides a container
- `text-left`: This left justifies text in its container
- `text-right`: This right justifies text in its container
- `text-center`: This center aligns text in its container
- `text-justify`: It left and right justifies text in its container

Rhythm

For most of us, the overall rhythm in Foundation is good enough. The average person will not notice any flaws, but to those who pride themselves on designing websites with a perfect **vertical rhythm**, they would say that Foundation is lacking. With the grid system, font sizing, and the standard way of implementing vertical spacing, Foundation-based websites look professional, and that's what matters.

In its style sheet, Foundation uses a mix of **px**, **percentage**, and **em** units of measure. It uses px for media queries, borders, and components such as forms and menus, where precise control is needed. Percentage is used for relative widths (for example, row and column width). Font size, margins and padding, and line height are specified in em units. These are the things normally associated with typography.

You can check the website or the Foundation CSS style sheet to see the default styles that Foundation applies to regular text, headings, and other HTML elements.

One thing that is both an asset and a liability is the use of em units for font sizing. When you change a font to a size other than the default size for a block, specifications such as line height and margins adapt proportionally. For the most part, that's a good thing because everything within the block scales proportionally. But its downside is that you sometimes don't want something to scale.

A simple example of this is the bottom margin. When you use a smaller font and the margin between paragraphs shrinks, that's good. But Foundation uses margins below elements to space things on a page. So now the margin below the last paragraph is going to be smaller than that of other blocks, and that throws off your overall vertical spacing.

Be aware of the measure types used, and you'll be able to quickly identify the cause of something not appearing as you might hope.

Common classes

Foundation supports some common classes that are used in several components. These include size, color, and radius classes. The following is a summary:

Size	Color	Radius
`tiny`	`primary`	`square`
`small`	`secondary`	`radius`
`medium`	`alert`	`round`
`large`	`success`	

The size classes will adjust the overall size of an element. `Medium` is the default size if no size style is specified. The `small` and `large` classes here have nothing to do with the `small-n` and `large-n` classes that are used to lay out a grid.

The color classes can be used to indicate some meaning to the viewer of your website. When you download a custom installation of Foundation, you can choose your own colors for each of these classes. For those elements that support these colors, the `primary` color is the default and will be used unless you override it with one of the others.

The radius classes refers to the corners of an element. The default is `square` and it will be in effect if nothing is specified. The `radius` class gives an element a slightly rounded corner. You can change the amount of rounding for this class when you download a custom installation of Foundation. The `round` class yields a much greater rounding on the corners.

You can choose one class from each column that is supported by the element where you want to apply these classes. We'll cover some examples as we explore various components.

Visibility

One of the powerful components of Foundation is its visibility classes. When you want to display something on a small device but not on a large device, or vice versa, you can use visibility classes. When you want your page to display differently for landscape versus portrait, you can use visibility classes. Foundation even includes the ability to identify touch-enabled devices.

Let's first look at device width. There are four device width breakpoints: small (less than 768 px), medium (768 px to 1279 px), large (1280 px to 1439 px), and x-large (1440 px and greater). The px values here are what you get when you install Foundation defaults.

When we talked about the grid system, we had small and large devices. Now we have four device widths to consider. Small happens to be the same in both situations, but large in the grid context covers medium, large, and x-large in the visibility context.

For device orientation, there are two categories: landscape and portrait. For touch, a device is either identified as a touch device or it isn't. There is only one category.

Foundation uses media queries to determine a device width and its orientation. To determine whether a device is touch-enabled, it uses JavaScript from `modernizr.js` that you should have loaded at the top of your web page.

There are two actions that you can perform with each of these categories: show or hide. When you put a `show-for-[category]` class on a container and that category is true, the content within that container may be displayed. When you put a `hide-for-[category]` class on a container and that category is true, the content within that container will be hidden.

The following is a simple example:

```
<img class="show-for-small" src="./images/little.png"/>
<img class="hide-for-small" src="./images/big.png />
```

The first `<img>` tag shows `little.png` only on small devices, while the second `<img>` tag shows `big.png` for all devices except small ones. From a visibility perspective, this works well. From a performance perspective, both images always get loaded into the browser even though only one gets displayed.

While working with the grid system, these two are the only ones you need to know in order to display one thing on small devices and something else on all others.

However, when you want finer control or need to make a distinction for orientation or touch, there are more classes to choose from, including ones such as `medium-up` or `large-down`. Those encompass anything larger (`-up`) or anything smaller (`-down`).

Use the visibility classes to limit where content will be displayed. Not specifying means that content will be visible unless there is some other limiting class.

For example, if we specify `show-for-small` and `hide-for-portrait` on a container, content within that will be visible on any small device as long as it is not held in portrait orientation. Also, any device other than a small one or any device held in portrait orientation will have the content to be hidden. Since touch is not specified, it has no effect and therefore content may be displayed whether the device is touch enabled or not.

Starting with simple components

In this section, we'll describe a few of the simpler components, or widgets if you prefer that term. Then we'll move on to more complex components. This section will also demonstrate the use of the common color, size, and radius classes.

These components may or may not be standalone components on Foundation's download page. If not, they can be found inside a component that is present.

Keystrokes

The HTML `<kbd>` element is intended for representing user input—keystrokes. Foundation adds some custom styles to this element and adds a `keystroke` class that can be applied to any inline element to get the same effect.

Each of these will have the same effect:

```
<p>Press <kbd>Enter</kbd> to continue ...</p>
<p>Press <span class="keystroke">Enter</span> to continue ...</p>
```

If you look closely, you'll see that the word **Enter** is highlighted and displayed in a monospace font:

Press `Enter` to continue ...

Inline lists

Foundation includes a simple `inline-list` class to present a list as a horizontal inline list. The following is an example code:

```
<ul class="inline-list">
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
  <li>Item 4</li>
</ul>
```

This will produce a list as shown in the following screenshot:

Item 1 Item 2 Item 3 Item 4

Add `<a>` links to these items, and you have a basic horizontal menu.

Tables

While we use the grid system for layout, tables still have a role to play, that is, to display tabular data. To avoid the terms, rows, and columns (which are layout terms), while describing tables, we'll use tuples and attributes.

The HTML elements that Foundation supports include: `<table>`, `<thead>`, `<tbody>`, `<tfoot>`, `<tr>`, and `<td>`.

The following screenshot is what a table with three tuples and three attributes, with a header and footer, will look like when Foundation's table styles are applied:

Attribute 1	Attribute 2	Attribute 3
tuple 1 attribute 1	tuple 1 attribute 2	tuple 1 attribute 3
tuple 2 attribute 1	tuple 2 attribute 2	tuple 2 attribute 3
tuple 3 attribute 1	tuple 3 attribute 2	tuple 3 attribute 3
Summary 1	**Summary 2**	**Summary 3**

Foundation takes care of the presentation. That includes a table border, highlight, and bold for the attribute headers and footers, and if you look closely, you'll see that every other tuple, tuple 2 in this example, has a shaded background.

You can use tables and the visibility classes together. Foundation has made some special considerations for tables in the visibility component.

Labels

The labels component is included in the typography group on Foundation's download page. Labels support the color and radius common classes, but not the size classes.

The following are the color labels:

```
<div class="small-12 columns">
  <span class="primary label">Primary</span>
  <span class="secondary label">Secondary</span>
  <span class="alert label">Alert</span>
  <span class="success label">Success</span>
</div>
```

The following screenshot is what the labels look like with Foundation's color styles:

Foundation uses a white font for the colored labels and a dark font for the `secondary` label with a light background. The `primary` class is more for documentation purposes. It is not specifically defined in the style sheet.

The following code shows labels with a radius class:

```
<div class="small-12 columns">
  <span class="square label">Square</span>
  <span class="radius label">Radius</span>
  <span class="round label">Round</span>
</div>
```

The following screenshot is what the labels look like with Foundation's radius styles:

The `square` class is for documentation purposes only. If you use it, you'll get square corners, but it is not actually defined in the style sheet.

vCard and vEvent

A handy little widget that Foundation introduces in the typography component is the vCard. A vCard is an electronic standard for business cards.

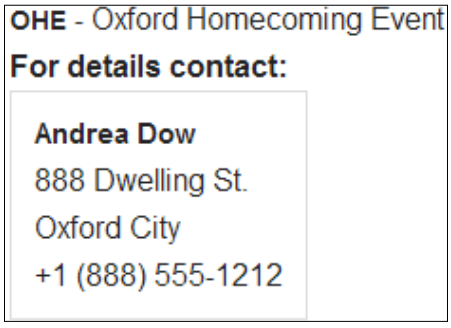
Create an unordered list with the `vcard` class and list the items you want to present. The only subclass given special consideration is `fn` for full name. If you want custom presentation for other attributes, such as street or e-mail addresses, you will need to add them to your custom style sheet.

In addition to the `vcard` class, Foundation also includes a `vevent` class with a couple of options: the `<abbr>` tag and the `summary` class.

The following is an example that demonstrates the vEvent and vCard components:

```
<div class="vevent">
  <p><abbr>OHE</abbr> - Oxford Homecoming Event</p>
  <p class="summary">For details contact:
  <ul class="vcard">
    <li class="fn">Andrea Dow</li>
    <li class="street">888 Dwelling St.</li>
    <li class="locality">Oxford City</li>
    <li class="phone">+1 (888) 555-1212</li>
  </ul></p>
</div>
```

The preceding code produces the following output when Foundation's styles are applied:



In this example, Foundation has styles for the vEvent abbreviation and summary and the vCard full name. The other classes in the vCard list have not been given any special styling. That is left to us. The vCard itself is placed within a box with a 1 px border.

Navigation

Foundation includes some basic navigation components that rely only on style sheets and do not require JavaScript to function. These include breadcrumbs, pagination, side nav, and sub nav. You can craft these by hand in your web pages or you may find them integrated into a content management system that uses Foundation.

Breadcrumbs

Breadcrumbs are often used in websites to show the context of a page. Where is the current page in the website hierarchy? The context could represent a menu structure, a category structure, or some other hierarchy. You decide.

There are no HTML elements that must be used for breadcrumbs. You can use nested `<div>` elements, an unordered list, or a `<nav>` element with nested `<a>` links. They all work. What matters is that the outer container has the `breadcrumbs` class and the nested elements include the breadcrumb content.

There are a couple of classes that can be applied to the nested elements: `current` and `unavailable`. Each one will disable any links associated with that element. The `current` element gets a dark gray font color, and the `unavailable` element gets a light gray font.

Let's put this all together in an example as follows:

```
<nav class="breadcrumbs">
  <a href="#">Home</a>
  <a href="#">Features</a>
  <a class="unavailable" href="#">Security</a>
  <a class="current" href="#">Password</a>
</nav>
```

When viewed through Foundation's styles, it looks like the following screenshot:

Foundation changes all the text to uppercase. The **HOME** and **FEATURES** items each have active `<a>` links. The **SECURITY** item would normally be a link, but we've made it unavailable. **PASSWORD** represents the current page.

Pagination

Use the pagination component when you want to show where you are in a sequence of pages. News or blog sites often break a story into small chunks and spread it over several web pages. Perhaps you have a long list of items that you want to index by date or alphabetically. Whatever the reason, pagination helps track a page within its sequence.

This component does require an unordered list to structure the content, so it isn't as flexible as the breadcrumbs component in this respect. Just add the `pagination` class to the `<ul>` element to implement this component.

It supports the `current` and `unavailable` classes with similar effect. The main difference is that the `current` class sets the background of the current element to the primary color and the font to white. This makes the current page clearly stand out.

Another optional class is `pagination-centered`. Use it on a container holding the pagination element to center the whole thing.

The following is an example:

```
<div class="small-12 pagination-centered column">
  <ul class="pagination">
    <li><a href="">&laquo;</a></li>
    <li><a href="">A</a></li>
    <li class="current"><a href="">B</a></li>
    <li><a href="">C</a></li>
    <li class="unavailable"><a href="">&hellip;</a></li>
    <li><a href="">Z</a></li>
    <li class=""><a href="">&raquo;</a></li>
  </ul>
</div>
```

The preceding code results in a pagination widget that looks like the following screenshot:

We used the `<ul>` and `<li>` tags to structure the pagination elements. We packaged the whole thing in a column and added the `pagination-centered` class.

You can decide exactly what you want to include in each item of your list. Here we've included left and right arrows at the beginning and the end. They need links to take a reader back or forward one page in the sequence. The current page, **B**, is highlighted with a blue background. And because we didn't include all the letters in the alphabet, we put some ellipses between **C** and **Z** and gave it an unavailable class so that viewers can recognize that it isn't clickable.

Side nav

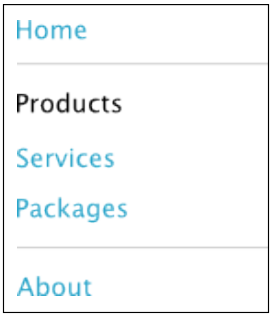
The side nav component provides a simple one-level vertical menu. Like the pagination component, it requires that you use an unordered list. The `<ul>` element is given the `side-nav` class, and the `<li>` elements become the items in the menu. When you want a horizontal divider in your menu, you can add an `<li>` element with the `divider` class. You can add the `active` class to indicate an item that represents the active link.

Side nav will use the full width available to it, so you'll want to place it inside a column or another container that confines it.

The following is an example:

```
<div class="small-2 column">
  <ul class="side-nav">
    <li><a href="#">Home</a></li>
    <li class="divider"></li>
    <li class="active"><a href="#">Products</a></li>
    <li><a href="#">Services</a></li>
    <li><a href="#">Packages</a></li>
    <li class="divider"></li>
    <li><a href="#">Company</a></li>
  </ul>
</div>
```

The preceding code produces the following vertical navigation:



We used an unordered list. It is inside a `small-2 column <div>` element, which contains it. The **Products** item was given the `active` class so it displays as black text while all the other items are blue, indicating that they have a link associated with them. And we tossed in a couple of dividers to separate the menu into logical groups.

Sub nav

The sub nav component provides a simple one-level horizontal menu. It requires that you use a data list. The `<dl>` element is given the `sub-nav` class, and the `<dt>` elements become the items in the menu.

There is no divider like there is in the side nav. You can add the `active` class to indicate an item that represents the active link. This will highlight the active link with the background of the primary color, white text, and rounded edges.

The following is the example of the side nav converted to a sub nav component:

```
<div class="small-6 column">
  <dl class="sub-nav">
    <dt><a href="#">Home</a></dt>
    <dt class="active"><a href="#">Products</a></dt>
    <dt><a href="#">Services</a></dt>
    <dt><a href="#">Packages</a></dt>
    <dt><a href="#">Company</a></dt>
  </dl>
</div>
```

The preceding code produces a horizontal menu that looks similar to the following figure:

The menu links are a light gray, not the default link color. The active link is a rounded label with the primary color as the background.

Buttons

Buttons can be big, small, colored, rounded, dropdown, split, and grouped. You need a button on your website? Foundation has the classes for it in its button components.

Remember labels? A button is similar to a label with an action behind it.

Regular buttons

Foundation supports the `<button>` element and the `button` class. Most of the time, you can use one or the other, and it will have the same effect.

Other than the text in the button, the following will be identical:

```
<button>&lt;button&gt;</button>
<div class="button">&lt;div class="button"&gt;</div>
<span class="button">&lt;span class="button"&gt;</span>
<a href="#" class="button">&lt;a class="button"&gt;</a>
```

The following is the screenshot resulting from the preceding code:

Compared to labels, buttons have a border effect that makes them appear raised. The default button is taller than a label. It has special hover and focus effects. Buttons support the color and the radius styles. They work in the same way as they do for labels. Buttons also support the `size` classes. You can choose one class from each category (`size`, `color`, and `radius`) and combine them to get a unique button.

There are a few additional classes available to tweak your buttons:

Class	Description
`left-align`	It left-justifies the text in the button. Typically it is centered.
`right-align`	It right-justifies the text in the button.
`expand`	It expands the button horizontally to fill the container the button is in.
`disabled`	It overrides the hover and the focus effect. It disables any link associated with the button and makes it a lighter shade so that it appears disabled.

Drop-down buttons

Foundation includes some extra styles for drop-down buttons. These add a down arrow to the right of the text on a button. There are two types: `dropdown` and `split`. The `dropdown` class simply adds the down arrow. The `split` class is a drop-down with a visual split in the button. The normal button text is to the left, and the drop-down arrow is to the right.

The syntax for the split button requires the addition of empty `<span>` tags within the element. Here is an example of each:

```
<div class="dropdown button">dropdown</div>
<div class="split button">split<span></span></div>
```

The following screenshot is how these appear on the screen:

Unlike regular buttons, these ones require the `button` class. They will work with the `<button>` element only if it is also given the `button` class. They support all the size, color, and radius styles, as well as other tweaks for regular buttons.

Button groups

You may have noticed from our examples that buttons will line up beside each other as long as there is space available. You could use this feature to build a crude menu simply by placing buttons together.

Foundation improves on this with the use of button groups. It then takes that a step further by allowing you to combine button groups onto a button bar. Now your crude menu will start to look professional.

To get the full effect, you will need to use an unordered list for the buttons in a button group. Each will stretch as much as needed, first horizontally and then vertically, to fit the text you have inside the button.

To make the buttons of equal width, independent of the text, use the `even-n` classes. These spread the buttons evenly over *n* spaces to fill the horizontal space available.

The following is a default button group classes that is spread evenly over three equal spaces:

```
<ul class="button-group">
  <li><a href="#" class="button">button</a></li>
  <li><a href="#" class="button">button button</a></li>
  <li><a href="#" class="button">button button button</a></li>
</ul>
<ul class="even-3 button-group">
  <li><a href="#" class="button">button</a></li>
  <li><a href="#" class="button">button button</a></li>
  <li><a href="#" class="button">button button button</a></li>
</ul>
```

These two button groups look as shown in the following screenshot:

The size, color, and radius styles work on buttons in a button group. Add the radius styles at the group level, and Foundation will take care of applying the rounded corners only to the outside corners of the first and last button in the group.

The following is an example of the round class:

```
<ul class="even-3 round button-group">
```

The preceding code will appear as shown in the following screenshot:

A button bar can contain more than one button group. All the styles for button groups can be applied, and Foundation keeps the groups apart. Just put the button groups together inside a container with the `button-bar` class. The following is an example:

```
<div class="button-bar">
  <ul class="radius button-group">
    <li><a href="#" class="small button">Button A</a></li>
    <li><a href="#" class="small button">Button B</a></li>
    <li><a href="#" class="small button">Button C</a></li>
  </ul>
  <ul class="radius button-group">
    <li><a href="#" class="small button">Button X</a></li>
    <li><a href="#" class="small button">Button Y</a></li>
    <li><a href="#" class="small button">Button Z</a></li>
  </ul>
</div>
```

The following screenshot shows this button bar with two button groups:

Now we have two groups of buttons on one button bar separated by a gutter. Even the `radius` class works as we would hope and rounds only the outside corners.

Block grids

Block grids are a special type of grid structure that you can use when you want to space content items evenly across the space you are working in.

You can have up to twelve items. This is independent of the number of cells you choose as the basis for your grid, if you do a custom download of Foundation. You will have at most twelve items even if you download a custom installation of Foundation.

Block grids use HTML lists with a special `block-grid-n` class. It is generally used within a content area of your page. The following is a simple example:

```
<ul class="small-block-grid-5">
  <li> <!-- content 1 --> </li>
  <li> <!-- content 2 --> </li>
  <li> <!-- content 3 --> </li>
  <li> <!-- content 4 --> </li>
  <li> <!-- content 5 --> </li>
</ul>
```

This code snippet could be used anywhere you want to display these five items spread evenly across whatever container they are in. The following screenshot shows how the preceding code appears:

The default gutter size for block grids is 20 px instead of 30 px, and there is a built-in horizontal gutter, which shows up below your content.

Like column definitions, you can specify the `small-n` or `large-n` classes. The `small-n` class alone will be in effect for small devices and regular displays. The `large-n` class alone will be in effect for regular displays, while small devices will display the content as though there was no block grid. Using `small-n` and `large-n` together, you can specify the number of items to display on small devices separately from regular displays.

One of the features of block grids is that you can have any number of items to display. Foundation will spread the number you specified in a block grid across your content area and then wrap the remaining onto the next line. It will keep doing this until all items are displayed.

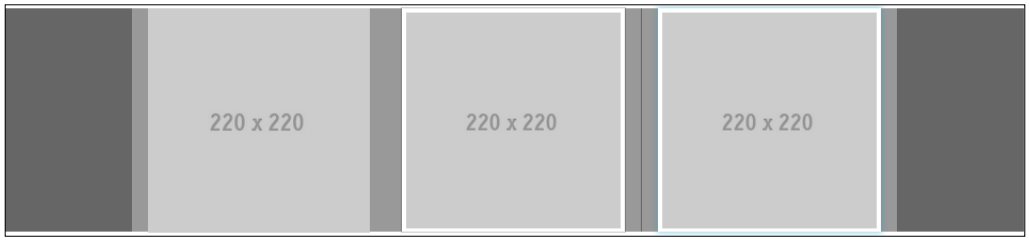
Thumbnails

The thumbnail component provides a simple mechanism to achieve a standard look-and-feel while attaching links to images. Just add the `th` class to your container element.

The following code is a row with three columns where each column has an identical image:

```
<div class="row">
  <div class="small-4 column">
    <a href="#"><img src="images/220x220.gif"></a>
  </div>
  <div class="small-4 column">
    <a href="#" class="th"><img src="images/220x220.gif"></a>
  </div>
  <div class="small-4 column">
    <a href="#" class="th"><img src="images/220x220.gif"></a>
  </div>
</div>
```

The output of the preceding code will look as follows:



The first image is plain. All the images are wrapped in a `<a>` link. Foundation adds a 4 px white border around the containers with a `th` class and a 1 px shadow around that. The third image has a mouse over it, so it is in the hover state. If you look carefully, you can see that it causes the border to turn blue and fuzzy.

One additional class that is supported is the `radius` class. As with buttons, it will make the borders have slightly rounded corners.

Panels

The panel component is used to highlight areas on your page. Panels are given a background color and a border that is 1 px wide.

There are two types of panels: regular and callout. The regular panel has a light gray background while the callout panel uses the primary color for the background. Font colors are adjusted to suit their respective backgrounds.

A subtle tweak that Foundation adds to panels is to ensure there is no top margin on the first element in a panel, or a bottom margin on the last element in the panel. At the same time, it adds padding all around the container.

Panels also allow for the `radius` class to round the corners slightly. To use panels, just add the `panel` class to a container. The following is an example:

```
<div class="row">
  <div class="small-4 column">
    <h1 class="subheader">No panel</h1>
    <p>Hi, I'm a typical column in Foundation. Notice that my
background color is the same as any column, in this case plain white.
And I have a no border around me.</p>
  </div>
  <div class="small-4 column panel">
    <h1 class="subheader">Regular</h1>
    <p>Hi, I'm a regular panel in Foundation. Notice that my
background color is light gray, #f2f2f2 to be precise. And I have a
thin border around me.</p>
  </div>
  <div class="small-4 column panel callout radius">
    <h1 class="subheader">Callout</h1>
    <p>Hi, I'm special. I take on the primary color as background,
#2ba6cb. My title and text is light and my thin border has slightly
rounded edges.</p>
  </div>
</div>
```

The preceding code results in the following in a browser:

Flex video

The flex video component is a slick little helper when you want to embed a video in your website. When you embed a YouTube video in a column without the flex video component, it isn't well contained by default and will spread out under the column beside it. Flex video packages it into the available space quite nicely.

It supports YouTube, Vimeo, and other streaming sites that use iFrame, embed, or object elements.

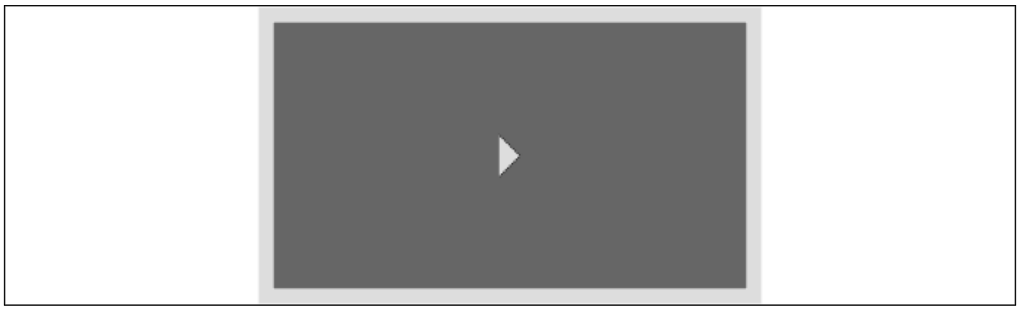
To use the flex video component, just create a element with the `flex-video` class. Assuming you are putting this inside a column, you need to create a new element inside that column. It doesn't work properly if you add the `flex-video` class to a column div.

The default video size will be a traditional 4:3 ratio. You can change that to 16:9 by adding the `widescreen` class. Because of the way Vimeo places the video controls, there is a special `vimeo` class, which tweaks the layout. This can be used with or without the `widescreen` class.

A nice enhancement is to use the panel component to frame the video inside a panel. The following code shows all of these features:

```
<div class="row"><div class="small-6 small-centered column panel>
  <div class="flex-video widescreen vimeo">
    <!-- embedded video -->
  </div>
</div>
```

The following screenshot shows how this example will appear:



The video uses half of the available space centered because we used the `small-6`, `small-centered`, and `column` classes within a row. There is a frame around the video as a result of the `panel` class. The `flex-video` class keeps the video within the column, but uses the full column width. The `widescreen` class sets the video dimensions to a 16 x 9 ratio. The `vimeo` class eliminates padding that would otherwise be above the video.

Pricing tables

Are you ready to make money with your website? Do you have something to sell? Foundation has a pricing table component to help you with that.

Foundation sets a unique style for each of the following classes: `title`, `price`, `description`, `bullet-item`, and `cta-button` (call to action).

The setup is quite simple. Add the `pricing-table` class to an outer container for the pricing table itself with inner containers for each attribute. The presumption is that you'll use an unordered list, but that isn't necessary. Nested `<div>` elements work just fine.

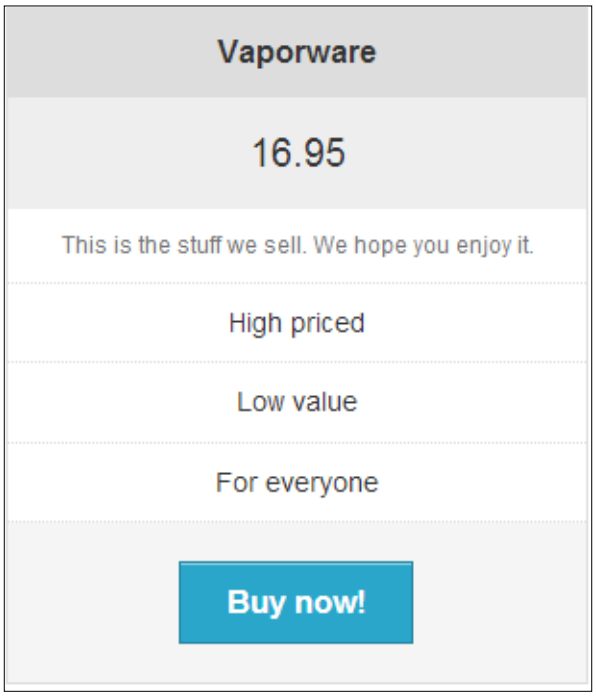
You still need to include numeric formatting in your price. You also need to wrap your `cta-button` in an `<a>` element in order to provide a link when a user clicks on it. You can make the entire `cta-button` container clickable by wrapping it in an `<a>` element. Or you could make your `cta-button` an actual button by adding the `button` class.

The following code incorporates these things:

```
<ul class="pricing-table">
  <li class="title">Vaporware</li>
  <li class="price">16.95</li>
```

```
  <li class="description">This is the stuff we sell. We hope you enjoy
it.</li>
  <li class="bullet-item">High priced</li>
  <li class="bullet-item">Low value</li>
  <li class="bullet-item">For everyone</li>
  <li class="cta-button"><a href="#" class="button">Buy now!</a></li>
</ul>
```

The following screenshot shows how the preceding code will be displayed:



Progress bars

The progress bar component gives you a way to visually show a measure of completeness for some task or activity. It is useful as a single snapshot of progress. If you want it to update in real time as a task progresses, you will need to find a way to do that as there is no built-in JavaScript to do that.

This component includes two new classes: `progress` and `meter`. Use `progress` as the container and `meter` for the shaded bar inside it. For simplicity, the following is the progress bar's basic form:

```
<div class="progress"><span class="meter"></span></div>
```

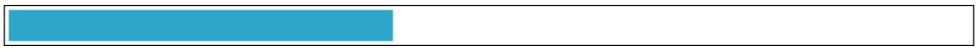
The following is the output generated from the preceding code:

The progress `<div>` is a container with a 1 px light border. The meter is the primary color. In the absence of a width specification, 100 percent is assumed, and each of these will fill their respective containers.

To show progress of less than 100 percent, set a width on the meter span as follows:

```
<div class="progress"><span class="meter" style="width: 40%"></span></
div>
```

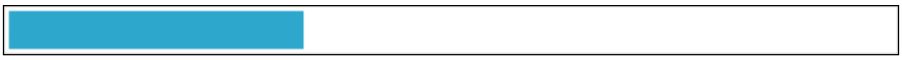
The preceding code displays a progress bar of 40 percent of the full container. That was the width set for the element with the `meter` class:



We can also use Foundation's built-in `small-n` or `large-n` classes for the progress div, the meter span, or both. The following is an example using both:

```
<div class="small-9 progress"><span class="small-4 meter"></span></
div>
```

The progress container has a width of `small-9` (75 percent) and the progress meter has a width of `small-4` (33.3 percent). The following screenshot is what this looks like:



Foundation's common color and radius styles are also supported. To use them, put the color and radius styles on the outer progress container, not on the inner progress meter.

Summary

That's it. Now you've seen all of Foundation's components that work without JavaScript. Pick the ones that best meet your needs. While JavaScript is a powerful ally, your website is much easier to build and maintain when you can develop with style sheets.

We studied the effect of using Foundation's typography. We saw how to use the visibility classes to show and hide content in different environments. We touched on several miscellaneous components that are useful styling aids. We saw how to use the navigation components that don't require JavaScript. We looked at all the ways to style buttons. And we looked at several more complex, special purpose components including block grids, thumbnails, panels, flex video, pricing tables, and progress bars.

Having mastered components that don't require JavaScript, now you are ready to move on to ones that do. In the next chapter, we'll study several more useful components called plugins. These do require JavaScript to function properly. The added capability and flexibility you get from these plugins make them well worth understanding.

3
Pick and Choose JavaScript Plugins

In the previous chapter we looked at Foundation's CSS components. There are several other components available that also require JavaScript, which is software that runs in the browser. JavaScript plugins can do a lot to enhance the capability of your website. But, along with all the wonderful features, you also get more complexity.

Troubleshooting HTML markup and CSS problems can be time consuming. Resolving problems with JavaScript is a much greater challenge. Having said this, Foundation's plugins will do amazing things for you, provided you set them up properly. This chapter will help you with that.

Most problems with plugins are a result of styles or JavaScript code from other sources or from combining the plugins in ways in which they haven't been tested. When you run into trouble, and you've checked and double-checked that the setup is correct, try isolating the plugin by setting it up in a simpler environment, or by removing extraneous style sheets and JavaScript files.

The plugins we'll cover in this chapter are Alerts, Tooltips, Dropdown, Sections, Top Bar, Magellan, Joyride, Orbit, Clearing, Reveal, Interchange, Forms, and Abide.

JavaScript installation

Foundation's JavaScript plugins can be installed at the top or bottom of a page. However, it is recommended that they be installed at the bottom; this will cause less delay in displaying the page content.

The simplest way to install JavaScript plugins is to install the complete Foundation package or choose plugins selectively in the custom download. Either way, use the JavaScript code from the bottom of the `index.html` file that comes with the download as your model. Copy the JavaScript code from that file to your page—change only the file locations if necessary—and you should be good to go.

To get started, you can load the entire JavaScript we have just described. But when you are going into production with a website, it is wise to load only the files you actually use.

Let's look at the way Foundation loads its JavaScript files. Using your favorite text editor, open the `index.html` file from a default installation of Foundation. The default installation will include all of the JavaScript plugins. Near the bottom, you'll see all of the JavaScript code for loading plugins, some of which will be commented out.

The first JavaScript code you'll see is for loading the jQuery library. If you are unfamiliar with jQuery, it is an open source library of JavaScript functions, which many websites today rely on. The Foundation toolkit also relies on jQuery, and Zurb includes a copy in its download.

Making things a little more complex in the Foundation JavaScript setup is the `zepto.js` library. It is a jQuery-compatible library. As it contains a subset of jQuery functions, it is lighter and therefore loads faster than jQuery. Using the code that Foundation provides, `zepto.js` will load if the browser supports it, otherwise, jQuery will be loaded.

You may find yourself in a situation where you always want jQuery and never load `zepto.js`. To do that, replace the script that loads zepto or jQuery with the following code, substituting `[location]` with the appropriate location for your installation:

```
<script src="[location]js/vendor/jquery.js"></script>
```

Next, in the JavaScript code, the `foundation.min.js` file is loaded. This file contains a minified version of all the JavaScript plugins that were included in the download. Minified means all of the extra whitespace and comments have been removed. The idea is to have one file that loads as quickly as possible.

When you are ready to selectively load plugins, disable (comment out) the script line that loads the minified JavaScript. At the same time, enable (uncomment) the script line that loads `foundation.js`. When you load selectively, the `foundation.js` file is required, as all the other plugin-specific files rely on it.

Lastly, you can enable the specific plugin(s) that you want to load. If you don't want a plugin, comment it out. If you do, make sure it is not commented out.

The last bit of JavaScript, just before the closing `</body>` tag, is where Foundation initializes all of the plugins. Always leave that code on the page.

If you are a JavaScript expert, you'll learn that you can do more things such as selectively initialize plugins, trigger events, and set options through JavaScript.

For the rest of this chapter, we'll look at each plugin and its setup on your web page. The name of the JavaScript file for each plugin is the same as the section title.

Alerts

The Alerts plugin offers a simple way to bring attention to a message. They are not pop ups, just inline content that is emphasized as a result of the classes you provide it with.

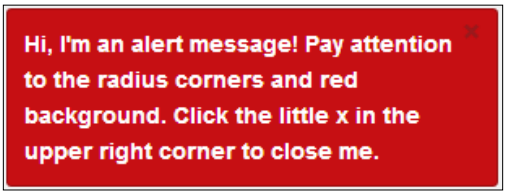
They support the color (`primary`, `secondary`, `alert`, and `success`) and radius (`square`, `radius`, and `round`) classes. As in the previous chapter, we use `primary` and `square` for documentation purposes only. Since they are not defined in Foundation's style sheets, we'll get default values when we use them and that's what we want.

The setup is very simple. Give a container the `alert-box` class along with any of the optional size or radius classes. If you want the alert message to stay open on the page and not have the capability of getting closed, your task is done. If you want the user to be able to close the alert message, add the `data-alert` attribute to the container and an `<a>` link inside the container just before the closing tag. The message alert will take the full width of its container.

We'll put all of this into an example as follows:

```
<p data-alert class="radius alert alert-box">
  Pay attention to the radius corners and red alert
    background. Click the little x in the upper right
    corner to close me.
  <a href="#" class="close">&times;</a>
</p>
```

On a computer screen, the preceding code will present a message box with the red alert color as the background. It will look like the following screenshot:



Tooltips

We've all seen tooltips, text that appears to jump out of a word or phrase when you hover over it on your screen. Many browsers support tooltips inherently. Add some text to the `title` attribute of an element, and you have a tooltip. Foundation adds styles and JavaScript to enhance tooltips.

Here is a basic tooltip:

```
<p>In this section we will explore Foundation's <span title=
  "Text that appears to jump out of a word or phrase when
  you hover over it on your screen.">tooltip</span> plugin.</p>
```

When you display this example paragraph on the screen, it will look normal. When you hover your mouse over the word **tooltip**, the text from the `title` attribute will display.

Add the `has-tip` class to the `<span>` tag that surrounds the word `tooltip`, and the text **tooltip** will be displayed as a darker color with a dashed underline. This indicates that the word holds a tooltip. Now when you hover over it, a question mark symbol will appear immediately and the tooltip itself will appear after a slight pause. All this was done using style sheets.

Next, add the `data-tooltip` attribute so that the JavaScript code can work its magic. This changes the look and behavior of the tooltip. It will now be black with white text in it, and it will be much bolder than before. When you first hover over the word, the question mark and the tooltip both pop up immediately.

The width of the tooltip has also changed. Instead of having the width of the containing element, it now spans to the right as far as it can go in the page. You can control the width of the tooltip by adding a value to the `data-width` attribute. The `data-width` attribute supports the usual width measurement units such as `px`, `em`, and `%`.

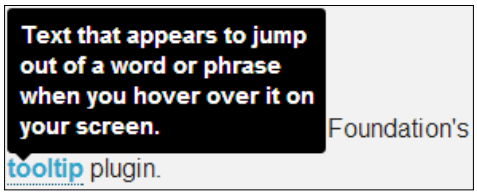
To change the location of the tooltip relative to its underlying element, add a positioning class to it: `tip-top`, `tip-right`, `tip-bottom`, or `tip-left`. These will force the tooltip to pop up above, to the right, below (default), or to the left of the element that holds the tooltip.

Like some other Foundation plugins, Tooltips support the `data-options` attribute, which can be used to disable tooltips for touch devices, that is, `data-options="disable-for-touch: true;"`.

Here is the code with an example of all these changes included:

```
<p>In this section we will explore Foundation's <span
  data-tooltip data-width="200px" data-options="
  disable-for-touch: true;" title="Text that appears to jump
  out of a word or phrase when you hover over it on your
  screen." class="has-tip tip-top">tooltip</span> plugin.</p>
```

This is what it looks like when you hover over the word **tooltip**:



The screen capture doesn't have the question mark that the mouse pointer turns into when hovered over the word **tooltip**. You can see that the tooltip itself stands out, and it is positioned above the word **tooltip** because we added the `tip-top` class. Because hovering doesn't work on many touch devices, we have disabled it for them.

Dropdown

Need a simple pop up that shows just below a target link? Foundation calls this a dropdown. It was intended initially as a simple drop-down list, but it now supports any content. Unlike the tooltip, which pops up when you hover over an element, with dropdown, you must click on the element to see the pop up.

The setup is straightforward. You have an item you want to attach a dropdown to, and you have the drop-down content. Put the item you want to attach a dropdown to in the `<a>` tag with a `data-dropdown` attribute and assign a value to it (an identifier). Build your content with an ID that is identical to the value of the `data-dropdown` attribute, give it the `f-dropdown` class, and add the `data-dropdown-content` attribute. The item you want to attach a dropdown to and the drop-down content are then linked by a common identifier.

Here is an example:

```
<a href="#" data-dropdown="menu-dropdown">About Us</a>
<ul id="menu-dropdown" class="f-dropdown"
  data-dropdown-content>
  <li><a href="#">Products</a></li>
  <li><a href="#">Services</a></li>
  <li><a href="#">Packages</a></li>
  <li><a href="#">Company</a></li>
</ul>
```

We put `#` as the link on the `<a>` element as a placeholder because Foundation is actually going to link to the drop down content.

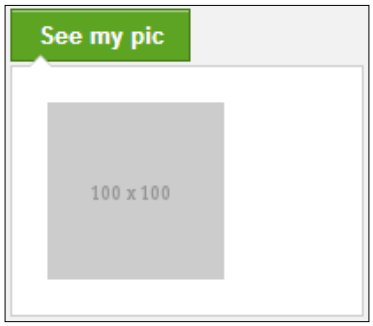
This is what it looks like when we click on the **About Us** link:

Since we have built each of the list items as a link, we have a simple drop-down menu. If you look closely, the **Packages** item is highlighted. That's because we have our mouse hovered over it. This example has used the default styling. You can customize the `f-dropdown` class or create your own class.

The Dropdown plugin is not restricted to lists. It can contain any content. Let's generate a dropdown with a simple image:

```
<a href="#" data-dropdown="content-dropdown" class=
  "small success button">See my pic</a>
<div id="content-dropdown" class="f-dropdown content"
  data-dropdown-content>
  <img src="./images/100x100.png" />
</div>
```

You can use any value you want to match the target link with the drop-down content. In our case, we have used `content-dropdown` because we are putting content in a dropdown. We also added the `content` class to the `<div>` tag that holds the image, so Foundation will style it differently than the `f-dropdown` class does on its own. Here is how the previous markup will appear when we click on the target:



We gave the target the `success button` classes because we wanted it to look better than plain text. Notice the padding around the image. That's a result of adding the `content` class.

Unless you override the styles, the default width of the dropdown will be 200 px. The Dropdown plugin supports the following size classes:

Class	Max width
`tiny`	200 px (default)
`small`	300 px
`medium`	500 px
`large`	800 px

Imagine the content you could put into an 800 px dropdown. Like all good things in Foundation, dropdowns are responsive, and they will use the width of their container up to the maximum width specified.

Section

The Section plugin is an all-in-one module that provides some great ways to organize content and navigation on your web pages. Rather than spreading your content over many pages, or making a very long page, you can arrange your content or navigation items into sections and display one section at a time.

Sections can be organized in horizontal tabs, vertical tabs, or in a collapsible accordion.

The setup is simple, especially given the power that you have with the Section plugin. There needs to be one container `<div>` with some sections inside it.

The outer container needs to have the `section-container` class. The type of container needs to be specified in two places: as an additional class on the container `<div>` and as the value of the `data-section` attribute. You can override the default settings using the `data-options` attribute.

Here are the types of containers you can choose from:

Container type	Description
`tabs`	Displays content tabs horizontally across the page.
`vertical-tabs`	Displays content tabs vertically down the page.
`horizontal-nav`	Displays navigation tabs horizontally across the page on regular displays and an accordion on small devices.
`vertical-nav`	Displays navigation tabs vertically down the page on regular displays, and an accordion on small devices.
`accordion`	Displays accordion bars vertically down the page. Used for navigation and content.
`auto`	Displays content tabs horizontally on regular displays and an accordion on small devices.

Each section is specified using the `<section>` tag or a `<div class="section">` tag. Within that, there needs to be a title and a content element. These can be any of the HTML elements: headings, paragraphs, divisions, and so on. Here is an example code for Section:

```
<section>
  <p class="title"><a href="#">Content 2</a></p>
  <div class="content">
    <img src="./images/content.png" alt="Content" />
  </div>
</section>
```

Each section must be comprised of a title and content element. These are identified using the `title` and `content` classes. You can put anything in the content area: text, images, forms, slides, and so on. You can even nest another section container if you want.

Let's see what each of the container types look like. First, we'll look at the content in horizontal tabs. The following is the code for it:

```
<div class="section-container tabs" data-section="tabs">
  <!-- Sections 1 to 4 -->
</div>
```

The sections will look like the following screenshot:

To present the exact same content with vertical tabs, replace the `tabs` class with `vertical-tabs`. Do this in two places in the previous code. Everything else remains the same.

Whether content areas are placed horizontally or vertically, they will make use of the full width of their container and will push other content down, the same way any HTML content does.

By default, the first section in the container will be visible when the web page loads. There will only be one section open at a time. When a user clicks on a tab, it will open and the currently open tab will close. Foundation does not provide a way to close all of the content areas for the horizontal and vertical tabs.

Here is the code for a horizontal navigation container:

```
<div class="section-container horizontal-nav"
  data-section="horizontal-nav" data-options="one_up: false;">
  <!-- Sections 1 to 3 -->
</div>
```

To get the exact same navigation to be presented vertically and to the left, replace `horizontal-nav` with `vertical-nav` in two places in the previous code. Everything else stays the same.

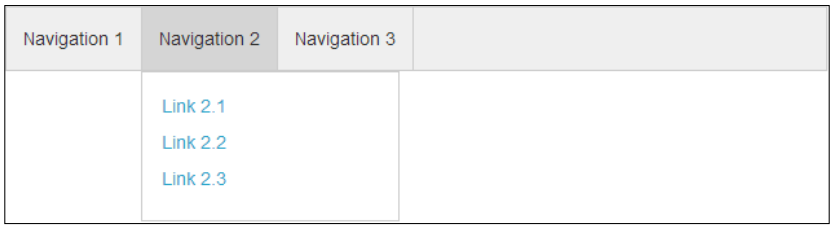
The navigation sections drop down or flyout over the surrounding content areas and they have a fixed width.

Here is an example section from the horizontal and vertical navigation containers:

```
<div class="section">
  <p class="title"><a href="#">Navigation 2</a></p>
  <div class="content">
    <ul class="side-nav">
      <li><a href="#">Link 2.1</a></li>
```

```
        <li><a href="#">Link 2.2</a></li>
        <li><a href="#">Link 2.3</a></li>
      </ul>
    </div>
  </div>
```

The following screenshot will be displayed upon executing the previous code:



These sections are the same as the content sections. Each section must have a title and content element, identified by the `title` and `content` classes. In these sections, we used a `<div>` element with the `section` class rather than the `<section>` element. You can use either of the syntaxes, and it will have the same effect.

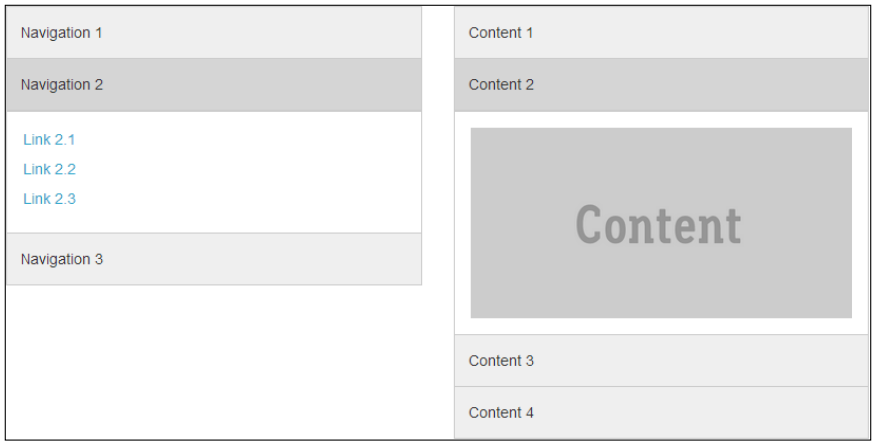
For styling, we used the `side-nav` class to handle the presentation of our links. This was done for convenience, and it requires that the navigation component be installed.

In these navigation containers, we set `one_up` to `false`. This overrides the default behavior of opening the first section when the page loads. No navigation item will be active and open until the user clicks on one of them.

The last unique type of container type to consider is the accordion, identified by the `accordion` class. It can be used as a content or navigation container. Here is the code for an accordion:

```
<div class="section-container accordion" data-section=
  "accordion" data-options="one_up: false;">
  <!-- Content or navigation sections -->
</div>
```

Here is how the accordion appears with the exact same sections we had in our previous examples:



In this screenshot, the part to the left has the same sections as our horizontal and vertical navigation examples, and the one to the right has the same sections as our horizontal and vertical tab examples. Each is shown with the same section open, as shown in the previous examples.

By setting `one_up` to `false`, all of the accordion sections will be inactive and invisible when the page loads. A user will need to click on one of the titles to open the corresponding content. The accordion allows the user to toggle a section between its opened and closed views, but only one section can be opened at a time. This applies to both the content and navigation containers.

The last container type we need to note is the default type, identified by the `auto` class. This results in a horizontal tab container for regular displays and an accordion for small devices. Actually, the auto, horizontal navigation, vertical navigation, and accordion containers are displayed as accordions on small devices.

Top Bar

We've covered several components and plugins that are intended for, or can be adapted to, navigation. Each of these is useful in its own way. Now we'll look at Foundation's most powerful navigation plugin, Top Bar.

It supports multi-level flyout menus, which work best as a fixed bar across the top of your web page. Also, it compresses to a vertical accordion style menu on small devices, and it can contain many different elements.

A Top Bar consists of an overall container with a title area and navigation section within it. Here is the high-level code:

```
<nav class="top-bar">
  <ul class="title-area"> ... </ul>
  <section class="top-bar-section"> ... </section>
</nav>
```

The outer container is identified with the `top-bar` class, the title section with the `title-area` class, and the navigation section with the `top-bar-section` class. Semantically, it makes sense to use the `<nav>`, `<ul>`, and `<section>` elements as shown in the previous code, but technically it works just as well when these are nested `<div>` elements.

In the title section, you can put a title name and an element to click on to open the accordion menu on small devices. Here is the markup:

```
<ul class="title-area">
  <li class="name">
    <h1><a href="./">Home</a></h1>
  </li>
  <li class="toggle-topbar menu-icon"><a href="#">
    <span>Menu</span></a></li>
</ul>
```

Always include an element with the `name` class. By default, it will show up on the left-hand side of the navigation bar on both small and large displays. You can add a link to the name. In this example, we used it as a home page menu item. You can also leave the name empty, but you need to keep it as a placeholder so the plugin works properly on small devices.

Use the elements as shown in the previous example, including the `<h1>` tag for the title, if you want to take advantage of Foundation's styling.

The second part of the title section is what gets displayed on small devices to open and close the vertical menu. The `toggle-topbar` class tells the plugin that this is the element that opens and closes the vertical menu on small devices. If you want Foundation's menu icon included, use the `menu-icon` class. Between the `<span>` tags, you can put any text to identify the menu. Omit if you don't want any text to be displayed. Include the `<a>` link as shown in the previous code, as it is needed to open and close the vertical menu on small devices.

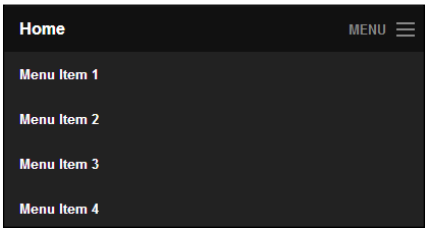
The navigation section is where you build your menu using unordered lists. Here is a simple, single-level list:

```
<section class="top-bar-section">
  <ul>
    <li><a href="#">Menu Item 1</a></li>
    <li><a href="#">Menu Item 2</a></li>
    <li><a href="#">Menu Item 3</a></li>
    <li><a href="#">Menu Item 4</a></li>
  </ul>
</section>
```

If we put together the code we have so far, including the outer container, the title section, and the navigation section, this is what it looks like in a regular display:

This is how it looks like on small devices before the menu is opened:

By clicking on **MENU** or the menu icon on the right-hand side, the menu will open up as shown in the following screenshot:



This is a basic Top Bar navigation. There are many variations and customizations that you can do to build on it.

The first thing is multi-level menu. Any of the `<li>` elements in our example can contain its own unordered list of menu items. Any one of those can contain another list, and so on. Foundation doesn't fix a limit to the number of menu levels, but the display width of your application soon will.

To make this work, add the `has-dropdown` class to the `<li>` element where you want the submenu. And add the `dropdown` class to the `<ul>` element that contains the submenu.

Another thing you can do is to have multiple lists at the top level. By adding the `left` or `right` class to these lists, you can control where they are positioned in the top bar, and you can control how they wrap when the browser is too narrow to display the whole thing in one line.

Adding the class `left` to the second list will cause that entire list to float down and to the left when the browser narrows. The first list is then above the second list. Adding the class `right` to the second list will cause that list to float to the right, and there will be a space between the two lists on wide views. When the browser is too narrow for both the lists, the second one will drop and start to slide under the first. You can add the `right` or `left` class to either of the lists and get different effects. Experiment and choose the combination that works best for you.

When you add the `right` class to a top-level list, Foundation changes the direction of the submenus. Instead of opening to the right (the default), they will open to the left. The idea is that this leaves more room in the browser for the submenus to display.

If you have two or more lists at the top level, as we just described, there might be two breakpoints. When squeezing your browser to test different widths, the first breakpoint will be when there is insufficient space to hold all the items at the top level, and one list wraps or slides under the other.

When you do a custom download of Foundation, you can choose a breakpoint for it to switch to the small device view. The default is 58.75 em (940 px). This is different from the point where two lists will break and slide one under the other.

You can add the `active` class to a `<li>` element in the menu to identify that it links to the current page. Foundation styles it differently, but you may want to customize it, as the default is so subtle that it's hard to notice.

If you want dividers to separate one menu item from another, you can add the `<li>` element with the `divider` class. Put this between every pair of menu items or just between logical groupings, whatever makes sense for you and your website.

You can put anything in the elements of your lists. Foundation provides some custom styling for buttons, labels, forms, and input boxes.

The best place to have the code for the Top Bar is at the top of the page, just after the opening `<body>` tag. When you do this, things just work. But you can put your Top Bar elsewhere on your page, even inside a container.

Foundation provides a couple of other options for positioning your Top Bar. You can constrain the content of your Top Bar to the maximum width of your grid. To accomplish this, wrap your `<nav>` element in a `<div>` element using the `contain-to-grid` class, as shown in the following code:

```
<div class="contain-to-grid">
  <nav class="top-bar">
    <!-- top bar sections -->
  </nav>
</div>
```

Constraining it this way only constrains the content. The background will still stretch to the full width of the browser. If you want to constrain the content and the background to the grid, put the Top Bar inside a 12-cell (full-row width) column instead.

You can fix your Top Bar at the top of the browser window and have it stay there as the user scrolls down the page. To accomplish this, wrap the `<nav>` element in a `<div>` element with the `fixed` class, as shown in the following code:

```
<div class="fixed">
  <nav class="top-bar">
    <!-- top bar sections -->
  </nav>
</div>
```

You can combine `fixed` with `contain-to-grid`. If you want the Top Bar inside a column until it reaches the top, where it fixes itself at the top rather than being scrolled out of sight, use the `contain-to-grid` and `sticky` classes.

One of the limitations of a touch device is that hovering isn't easy to support. To make your Top Bar navigation work better on touch devices without hovering, you can enable the click action instead of the hover action to navigate the flyout menus. Activate this by changing the setting via the `data-options` attribute, as shown in the following code:

```
<nav class="top-bar" data-options="is_hover:false">
```

Now a user will need to click on a menu item to trigger the dropdown or flyout. Hovering over the menu item will have no effect.

Magellan

The purpose of Magellan is to show the user where they are currently on a webpage. This is especially useful on long pages with many topics. Combine this with page bookmarks, and you can link to locations on a page as well as provide information to the user that they are in fact at a specific location.

Whether the user jumps to a location through a bookmark link, or scrolls down a page, Magellan will update and reflect the changing locations. To set it up, you need two things: a list of items to link to all the locations you want to mark, and a mark at each location.

The list can be nested HTML elements, including those from the sub nav, side nav, or button groups components. Foundation makes use of the `active` class from the sub nav component in Magellan. Foundation uses custom data attributes for the list of items that indicate locations and for the locations themselves. These include:

Data attribute	Description
`data-magellan-destination`	It is used to mark locations in a page: `data-magellan-destination="<named location>"`
`data-magellan-expedition`	It identifies the container that holds the list of indicators corresponding to the locations you have marked: `data-magellan-expedition="fixed"`
`data-magellan-arrival`	It identifies an indicator corresponding to a destination: `data-magellan-arrival="<named location>"`

There should be an arrival data attribute for each destination, and the value of `<named location>` for an arrival must be the same as the one for the destination location.

Let's put it together in an example. We need to mark the target locations and build a list corresponding to those locations. While not required, we've included the `<a>` links to the page locations so our Magellan list can be used to navigate to page locations as well as reflect where the user is on a page:

```
<!-- Our menu and Magellan location indicator -->
<div data-magellan-expedition="fixed">
  <dl class="sub-nav">
    <dt data-magellan-arrival="magellan-top" class="active">
      <a href="#top">Top</a>
    </dt>
```

```
      <dt data-magellan-arrival="magellan-schedule">
        <a href="#schedule">Schedule</a>
      </dt>
      <dt data-magellan-arrival="magellan-medical">
        <a href="#medical">Medical</a>
      </dt>
      <dt data-magellan-arrival="magellan-financial">
        <a href="#financial">Financial</a>
      </dt>
    </dl>
  </div>
  <!-- content areas -->
  <a name="top"></a>
  <h1 data-magellan-destination="magellan-top">Clinic
    Management</h1>
  <!-- content about clinic management in general -->
  <a name="schedule"></a>
  <h2 data-magellan-destination="magellan-schedule">Schedule</h2>
  <!-- content about scheduling -->
  <a name="medical"></a>
  <h2 data-magellan-destination="magellan-medical">Medical Records</h2>
  <!-- content about medical records -->
  <a name="financial"></a>
  <h2 data-magellan-destination="magellan-financial">Financial</h2>
  <!-- content about financial aspects -->
```

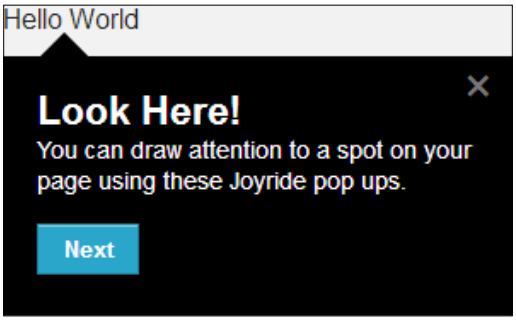
The menu/indicator works best at the top of the page, though it could be anywhere. As soon as you scroll down past the original menu/indicator location, Foundation will fix it at the top of the page. You can add the `active` class to the menu item that represents the starting point of the page so that the location is indicated when the page loads.

This is what the menu/indicator in the example looks like when the user is in the **Medical** area of the page:

Joyride

Would you like to guide the users through your page content? Joyride does that. It is another plugin that supports navigation within a page. It may be useful as a training tool or a guide where you want to describe an item on your page.

With default styles, there is a pop up with your message linked to a specific spot on your page. Here is what one looks like:



Hello World is the text we are drawing the attention of the user to. The default pop up is a block with a black background and white text. We specified the text **Next** as the button. Clicking on **Next** leads to the next pop up location in your sequence. The close icon in the upper-right corner will close the current pop up and end the sequence.

The preceding screenshot was taken from a regular display, where it had a default width of 300 px. On a small device, the pop up takes most of the complete width of the screen.

The setup is a little trickier than some plugins, but once you get it, you'll find it a worthwhile tool to have in your kit. There are three major steps:

1. Insert some HTML code in your page to identify each location where you want a pop up.
2. Build a list with the content of each pop up, linking each list item to one of the locations you identified in step 1.
3. Add some JavaScript to trigger the pop-up sequence when the page loads.

We'll go through the previous steps in more detail with examples.

1. First, identify a target location by adding an ID or a class to each container where you want a pop up to appear:

```
<div id="demoStop">Hello World</div>
```

2. Second, identify the list itself with the `data-joyride` attribute so that the JavaScript plugin can find it, and add the `joyride-list` class so that it will be hidden:

```
<ul class="joyride-list" data-joyride> ... </ul>
```

For each item in the list, add the `data-id` or `data-class` attribute with a value matching the ID or class at the location where the pop up is to appear. Include the text you want in the button that links to the next pop up, as the value of the `data-text` or `data-button` attribute.

```
<li data-id="demoStop" data-text="Next">
  <h4>Look Here!</h4>
  <p>You can draw attention to a spot on your page using
    these Joyride pop-ups.</p>
</li>
```

3. Third, insert the entire list at the bottom of your page just above the closing `</body>` tag. Start the pop up sequence when the page loads by adding the following JavaScript to the location where your plugins are loaded:

```
$(document).foundation('joyride', 'start');
```

You can get a simple pop up in the middle of the browser window simply by not associating a target location with an item in the Joyride list. Do not use the `data-id` or `data-class` attribute. For example:

```
<li data-text="OK">
  <h4>Closing</h4>
  <p>Thank-you for viewing this presentation.</p>
</li>
```

This would be a great way to present a sequence of slides in one place where the user could navigate forward from one pop up to the next by clicking on a forward-linking button.

The content of a pop up can be any valid HTML content. It could be a form, an image, text, or whatever you need. If you have a sequence of forms that you need the user to fill, Joyride can provide the mechanism for moving from one form to the next.

We've shown you enough to get started with Joyride and use it to build a simple intra-page navigation or sequence of slides. While we won't cover them here, there is more to explore in this plugin including the use of cookies, a timer, and the ability to override defaults with the `data-options` attribute.

Orbit

Orbit is a slide presentation plugin. By default, images are displayed one at a time, advancing automatically after a predetermined period. A user can stop and restart the presentation, click forward or backward, or go directly to a specific slide. It displays the slide number and count, and it displays a caption if provided.

While intended for images, Orbit can display any type of content (you can even use text). Orbit is similar to a PowerPoint presentation.

Here is the basic setup:

```
<ul data-orbit>
  <li><img src="./images/prep.jpg" /></li>
  <li><img src="./images/start.jpg" /></li>
  <li><img src="./images/cooked.jpg" /></li>
  <li><img src="./images/trek.jpg" /></li>
</ul>
```

That's it! We have an unordered list with the `data-orbit` attribute and content in each list item. From this simple setup, JavaScript takes over and presents the images as a slide presentation with controls and autoplay.

Let's add a fifth image with a caption so we have an example to look at:

```
<li>
  <img src="./images/OrbitSliderImage.gif" />
  <div class="orbit-caption">Orbit Slider Caption</div>
</li>
```

The caption is a `<div>` element with the `orbit-caption` class. It is placed inside the `<li>` element. It looks like the following screenshot:

In the upper-left corner, we have the slide number and slide count. That's just presentation. Nothing will happen if we click on it. In the upper-right corner, there is a bar that represents the time elapsed since the image appeared. It is about halfway in the previous screenshot. When it reaches the end, the next slide will replace the current image. To the right of the progress bar, there is a play icon that alternates between a pause and a play symbol.

Previous and next arrows are positioned at either side of the image. Clicking one of these will replace the image with the previous or the next image.

Overlaying the bottom of the image is the caption bar. That's the text we had put inside `<div class="orbit-caption">`, immediately following the code for the image.

Below the image we have five bullets, one for each slide. The bullet for the active slide is darker. A user can click on one of those bullets to go directly to the corresponding image.

There are a couple of things we can do to help load the slide presentation smoothly:

- Add a wrapper using the `slideshow-wrapper` class to prevent the display of content before JavaScript finishes applying styles
- Insert an empty `<div>` tag with the `preloader` class to display a rotating icon to indicate that the presentation is loading

Here is the code snippet:

```
<div class="slideshow-wrapper">
  <div class="preloader"></div>
  <!-- Slide presentation -->
</div>
```

There is also another more explicit way for your users to select a specific slide, deep linking. You first identify a list item with a name using the `data-orbit-slide` attribute. Then you link that item using the `<a>` tag with the `data-orbit-link` attribute.

For example, if we want to provide a direct link to the fifth image, we would add the `data-orbit-slide` attribute with a value to the target `<li>` element, as shown in the following code:

```
<li data-orbit-slide="orbit-slide-5">
  <img src="./images/OrbitSliderImage.gif" />
  <div class="orbit-caption">Orbit Slider Caption</div>
</li>
```

Then, to lead our user to the fifth slide, we add the `data-orbit-link` attribute with the same value to the `<a>` element, as shown in the following code:

```
<p>Click <a href="#" data-orbit-link="orbit-slide-5">here</a>
  to advance immediately to the last slide.</p>
```

Several default settings can be changed by adding the `data-options` attribute to the `<ul>` element in your slide presentation. Look at the Orbit JavaScript file, near the end, for a list of current settings.

For example, let's assume that you want to change the period of time that a slide displays before advancing on to the next slide. The default period is 10 seconds (10,000 milliseconds). Let's change that to 4 seconds (4000 milliseconds) by changing a setting in our `<ul>` element:

```
<ul data-orbit data-options="timer_speed:4000;">
```

Other things you can do with these settings include:

- Disable the slide effect when a new image appears
- Disable pause on hovering
- Disable navigation arrows, slide numbering, or bullets
- Enable variable-height slides
- Choose your own class names for many of the elements

Clearing

Remember when one of your friends posts a group of photos on Facebook, and you click on one of the thumbnails, the image behind that thumbnail pops up. You can navigate forward and backward to see more photos in that group. But you cannot do anything else on Facebook until you close the pop up. This is known as a **Lightbox**.

Foundation gives you the tools to build your own lightbox using the Clearing plugin. It is similar to Orbit in the way it presents images in a slideshow. But it is different in that it operates in a pop up (modal window), and it includes the thumbnails automatically to allow your user to select a photo to view.

The Clearing plugin itself doesn't style the thumbnails (border, hover, and so on.). But the thumbnail component can. Use it, or create your own styles if you prefer.

To set up a Clearing lightbox, add the `data-clearing` attribute to the `<ul>` element in an unordered list. Then add your images, full sized and thumbnails, to the `<li>` elements.

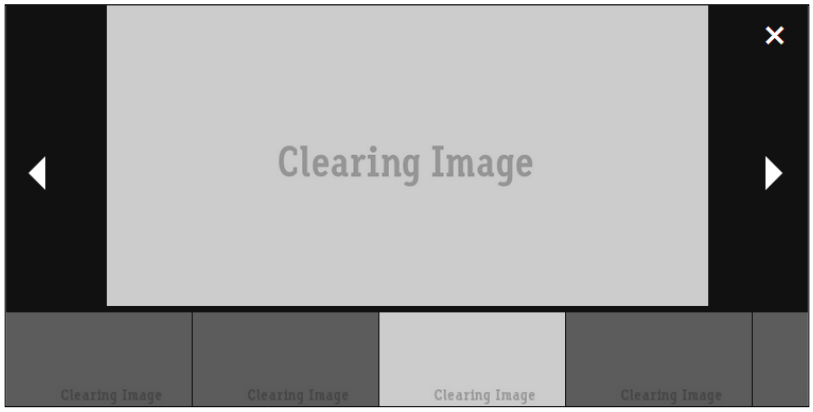
Here is an example:

```
<ul data-clearing>
  <li><a href="./images/prep.jpg" class="th">
    <img src="./images/prep-th.jpg"></a></li>
  <li><a href="./images/start.jpg" class="th">
    <img src="./images/start-th.jpg"></a></li>
  <li><a href="./images/cooked.jpg" class="th">
    <img src="./images/cooked-th.jpg"></a></li>
  <li><a href="./images/trek.jpg" class="th">
    <img src="./images/trek-th.jpg"></a></li>
</ul>
```

This will display the four thumbnail images side by side within the space we've allowed. If there isn't room for all the thumbnails, they will wrap onto the next line.

When your user clicks on any one of the thumbnails, that image will pop up in a modal window and the user can navigate forward, backward, or to another image in your list.

Here is a screenshot that shows what the Clearing pop up looks like:



In the previous screenshot, there are three controls. The close icon in the upper-right corner closes the pop up. The backward and forward arrows are used to move to the previous and the next images respectively. The thumbnails at the bottom provide a way to navigate to another image.

We said it was simple, but you do have to create your thumbnail images from your main images and load both of them onto the `<a>` links.

You can add a caption at the bottom of the image. It's the same idea as with the Orbit plugin but a different implementation. Add a `data-caption` attribute to the `<img>` tag:

```
<li><a href="./images/prep.jpg" class="th">
<img data-caption="Preparing chicken for grill"
src="./images/prep.jpg"></a></li>
```

Instead of displaying all of the thumbnails, you can display just one. This makes it easier to place your Clearing plugin in a sidebar or other small space. This is called the featured image. Add the `clearing-feature` class to the `<ul>` element in your setup, and the `clearing-featured-img` class to the `<li>` element that contains your preferred image.

Now your users will only see the featured image until they click on it. Then the modal window will pop up with the featured image displayed, and the user can navigate to other images in your list.

This plugin supports the `data-options` attribute, where you can change the settings in the JavaScript file, but there aren't as many settings to tweak as there were in Orbit.

Reveal

The Reveal plugin provides a pop up on which you can put anything you want to. It is similar to the Clearing plugin in that it puts the content on a pop-up modal window, but it is much more generic. An Orbit slider on a Reveal pop up makes a great presentation.

The setup is reasonably straightforward, but there are a few nuances that are not typical of Foundation plugins. The first of these is that the HTML specifying the Reveal modal needs to be the last HTML code on the page. It needs to be below all the other content but before the Foundation JavaScript at the bottom of the page. Here is what you have to do:

1. Add the `reveal-modal` class to a container that holds the content you want to include in the modal pop up.
2. Add a unique ID to that container.
3. Put that container just before your JavaScript, at the bottom of the page.
4. Build a link to trigger the pop up wherever you want it to appear on your page using the `data-reveal-id` attribute.
5. Optionally, add a control to close the pop up.

Here is a sample markup:

```
<p class="text-center">Click <a href="#" data-reveal-id=
  "orbitModal">here</a> to reveal a pop up with an orbit
  slider!</p>

<div id="orbitModal" class="reveal-modal">
  <ul data-orbit>
    <li><img src="./images/prep.jpg" /></li>
    <li><img src="./images/start.jpg" /></li>
    <li><img src="./images/cooked.jpg" /></li>
    <li><img src="./images/trek.jpg" /></li>
    <li><img src="./images/Orbit on Reveal.png" /></li>
  </ul>
  <a class="close-reveal-modal">&#215;</a>
</div>
<!-- Foundation JavaScript -->
</body>
```

The code with the `<a>` link to trigger the pop up can be anywhere in your HTML code. Include the data attribute `data-reveal-id` within the `<a>` tag to identify the specific Reveal container that you want to pop up.

In the container itself, add an `id` attribute with the same value that you gave to the `data-reveal-id` attribute. This is how Foundation matches them up. Add the `reveal-modal` class so that Foundation can create the pop up and apply the Reveal styles.

In the previous code, we have added a control to close the pop up in the `<a>` link with the `close-reveal-modal` class. You can put what you want for the text in that control. You could use the word `close`, the letter `X`, the multiplication character `&#215;` or whatever you choose.

Here is what the previous code will present after clicking on the link to trigger the pop up:

The text at the top in the previous screenshot is where we've provided a link to trigger the pop up. The pop up itself gets positioned a little below the top of the browser.

The only control related to the Reveal plugin is the little **x** in the top right corner. Click on that, or anywhere outside the modal, to close the pop up. All other controls and layout in this example are derived from the Orbit slider.

You can have more than one Reveal modal on a page as long as you give each one of them a unique ID so there is no confusion between them.

The Reveal modal supports size classes including `tiny` (30 percent), `small` (40 percent), `medium` (60 percent), `large` (70 percent) and `xlarge` (95 percent). The size classes adjust the percentage of the browser window used for the pop up and only works on regular displays, not on small devices. The default is 80 percent, and it doesn't correspond to any of the size classes.

Interchange

As a responsive toolkit, Foundation is based on the fact that by default all devices load the same content, and the only difference from one device to another is how the content is displayed. As one of the newest plugins, Interchange is designed for selective loading.

Instead of loading large resolution images to all the devices, you can load an image specific to a device. Especially on small devices, where users may be sensitive to bandwidth and processor performance, this makes it easier to optimize your web pages for those devices.

The setup is straightforward. Set an image in your `src` attribute as you normally would. Then add a `data-interchange` attribute where you specify alternate images and the conditions on which they will be used.

Let's look at an example:

```
<img src="./images/200x200.png" data-interchange=
  "[./images/400x400.png, (only screen and
  (min-width: 768px))]">
```

In this example, we specified one alternate image. You can have more than one alternate image as well. On devices less than 768-px wide, the smaller image (`200x200.png`) will be loaded. On everything else, the larger image (`400x400.png`) will be loaded.

The syntax for specifying alternate images is:

```
[<alternate image>, (<media query>)], [<alternate image>,
  (<media query>)] ...
```

Each alternate specification is enclosed in square brackets and contains an alternate image and a media query. Each media query is enclosed within parentheses. Commas separate an alternate image from its media query, and commas separate the alternate specifications as well.

Foundation includes some predefined media queries that you can refer to by name. For ones that are based on display width, they work similarly to the `small-n` and `large-n` classes in the grid system. Whatever is defined for the smallest width will remain in effect until a larger width takes over.

The named queries are as shown in the following table:

Name	Result
`default`	All display widths
`small`	768 px and above
`medium`	1280 px and above
`large`	1440 px and above
`landscape`	Landscape orientation
`portrait`	Portrait orientation
`retina`	High-density displays (for example, iPad 3)

To use these named queries, simply replace a verbose query with a name from the previous table. Our previous example will have the following code:

```
<img src="./images/200x200.png" data-interchange=
  "[./images/400x400.png, (small)]">
```

The image that you specify in the `src` attribute is the image that will be loaded by default, and it will always be loaded whether it gets replaced by an alternate image or not. From this perspective, it is best to specify the smallest image in the `src` attribute.

If you want to optimize and load only the image appropriate for the target, you can omit the image in the `src` attribute and specify all the alternatives in the `data-interchange` attribute. There are a couple of cautions with this approach: JavaScript must be running, and you must have an image for all possible views.

If you change the browser width after the initial load, Interchange will monitor that and load a different image to match the new width. However, it will not automatically load the default image in the `src` attribute after the initial page load. That should be specified in the `data-interchange` attribute whether or not it is specified in the `src` attribute.

Forms

Foundation brings style and class to forms (literally). Foundation has added custom styles to HTML form elements and then enhanced them with some unique classes.

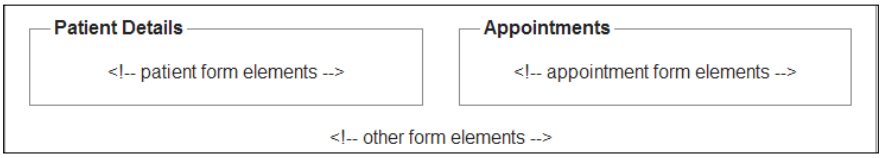
Much of this comes without the necessity of JavaScript, but it is there and is essential for some features. If you've mastered the grid system, you can put that knowledge to work. By their very nature, rows and columns make it easy to lay out a form. And Foundation takes the grid system even further by using its styles directly in form layouts.

Let's get started with an example of the high-level structure of a form:

```
<form><fieldset>
  <div class="row">
    <div class="small-6 column">
      <fieldset><legend>Patient Details</legend>
        <!-- patient form elements -->
      </fieldset>
    </div>
    <div class="small-6 column">
```

```
        <fieldset><legend>Appointments</legend>
          <!-- appointment form elements -->
        </fieldset>
      </div>
      <div class="small-12 column">
        <!-- other form elements -->
      </div>
    </div>
  </fieldset></form>
```

You start with opening and closing `<form>` tags. You fill it with form elements directly or organize your form elements into one or more groups using the `<fieldset>` tags. Foundation uses the `<fieldset>` tags to draw a thin line around the form elements inside those tags. If you include the `<legend>` tags, the enclosed text will be used to identify the group. This is what the previous code produces when it is displayed:



We used the grid system to lay out this form. In the previous screenshot, we had two `<div>` elements beside each other. Each of those had the `small-6 column` classes. Underneath those is another `<div>` element with the `small-12 column` classes. Each of the `small-6 column` containers have the form elements grouped using the `<fieldset>` tag, but the `small-12 column` container does not. We then grouped everything in the form into an aggregate `<fieldset>` tag. We didn't give it a title or `<legend>`, so Foundation draws a border but doesn't put any text on it.

There are several form elements you can choose from to build your form: labels, buttons, and input fields including text, the text area, checkboxes, radio buttons, switches, and dropdown selectors. Let's look at each one of these.

In the context of forms, labels are generally used to name or identify an input field. To use Foundation's styling for labels, put your label text inside the `<label>` tags. This is different from the label component we looked at in the previous chapter where the `label` class was used. For example:

```
<label>Enter name:</label>
```

Buttons are the same as they were in the button component, except that Foundation provides some additional styling for buttons used in forms. You can use the `<button>` tag or the `button` class.

The size, color, and radius classes work for buttons. If your button is at the beginning or at the end of another element, such as in an input field, you can add the `prefix` or `postfix` class so that Foundation properly styles only one end of the button with a radius class.

Input fields are where you capture the input from the user. You can use the `<input>` or `<textarea>` element, depending on your needs. Either of them will fill the width of their container. An `<input>` element will provide just one line of input whereas `<textarea>` will provide a vertical scroll bar when needed, and it will let the user stretch the input area.

You can add extra text before or after your input fields using the `prefix` and `postfix` classes. Like the input areas, these will use the full width of their container, so we can use the grid system to do the layout. They also support the radius classes, and Foundation will properly style only the outer corners.

Here is an example of an input field with a `prefix` text and a `postfix` button:

```
<div class="row collapse">
  <div class="small-2 columns">
    <span class="prefix radius">NAME</span>
  </div>
  <div class="small-8 columns">
    <input type="text" placeholder="surname, given name">
  </div>
  <div class="small-2 columns">
    <a href="#" class="button postfix radius">Done</a>
  </div>
</div>
```

This is how the previous code gets displayed:

The prefix text has radius border on the left-hand side and the postfix button has radius corners on the right-hand side. We used the `collapse` class to eliminate padding between the columns.

We also introduced the placeholder attribute in the `<input>` tag. As you can see, this code puts the placeholder text in the input field that gets replaced when the user enters the actual text. Be sure to include the `placeholder.js` file as it adds compatibility for older browsers.

Foundation uses the `type` attribute in the `<input>` tag to customize the input field.

Set up an `<input type="date">` element, and you should get a datepicker in the input field.

You can easily build checkboxes, radio buttons, drop-down selectors, and visual switches using Foundation's custom `form` classes. You enable these by adding the `custom` class to the `form` element:

```
<form class="custom"> <!-- form elements --> </form>
```

The simplest way to build a checkbox is to add the type `checkbox` to an `<input>` tag and wrap the whole thing in a `<label>` tag along with the label text you want. For example:

```
<label><input type="checkbox"> Neck</label>
```

Building a radio button is almost as easy. It is structured the same way, but it is given the input type `radio`. Assuming that you want more than one radio button to work together, you need to give the same name to the `<input>` tag for each radio button. Example:

```
<label><input name="myradio" type="radio"> Neck</label>
<label><input name="myradio" type="radio"> Back</label>
```

To any of the `<input>` tags in a checkbox or a radio button, you can add the attribute `checked` to set the default for that input as checked or the attribute `disabled` to disable that input. For example:

```
<label><input type="checkbox" checked disabled> Back</label>
```

To build a dropdown selector, use the `<select>` tag as the container, and use `<option>` tags to list the options you can choose from. For example:

```
<select class="small">
  <option>Neck</option>
  <option>Back</option>
  <option>Hip</option>
  <option>Knee</option>
</select>
```

The slider switch is a little more complicated, but it is a nice effect, and so it is worth the extra coding. Wrap the slider in a `<div>` tag with the `switch` class, add two radio buttons (Foundation will style them as sliders), add the `checked` attribute to the default radio button (make sure they have the same name), follow each `<input>` tag with a `<label>` tag, and add empty `<span>` tags just before closing the `<div>` tag.

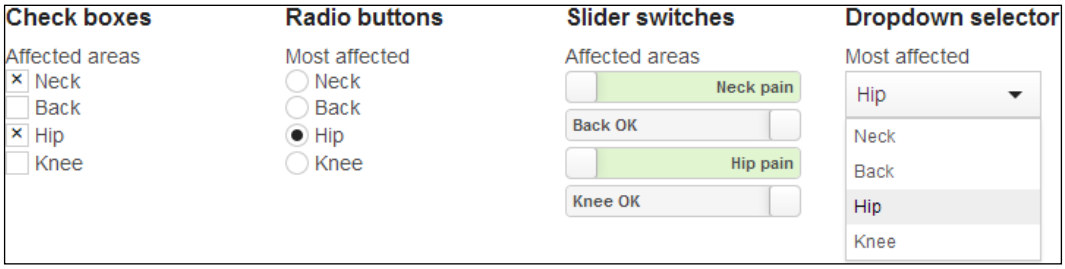
For example:

```
<div class="switch tiny radius">
  <input name="switch-hip" type="radio" checked><label>
    Hip OK</label>
  <input name="switch-hip" type="radio"><label>
    Hip pain</label>
  <span></span>
</div>
```

Typically, the two options of the slider would be labeled On and Off, True and False, Yes and No, or similar. But you can use any labels you want. We used `Hip OK` and `Hip pain`.

Foundation's size styles can be used in the drop-down selector, and both size and radius styles can be used in the slider switch.

Taking the code we used in these examples and wrapping each type in a column for presentation purposes, the following is what we'll see. On a computer screen, the background color of slider switches that are on will be light green while the background color of switches that are off will be light grey.



Using checkboxes and slider switches at a time, we can select as many options as we want. With radio buttons and the drop-down selector, we can select only one option at a time.

Building forms is a matter of putting the elements that we've discussed here, together in ways that make sense for your form. Much can be done with labels, buttons, prefix and postfix areas, input fields, and textareas. Adding custom form elements such as checkboxes, radio buttons, slider switches, and drop-down selectors gives you even more powerful forms.

Abide

Abide takes forms to a whole new level by providing field input validation. Fields can be marked as `required`, and they can be masked with a pattern that they must match. You can easily add error messages for incorrect entries.

Abide has its own JavaScript file in addition to the form's JavaScript file. Make sure you get both if you want to use Abide. When you install Abide, you only have to worry about JavaScript. Foundation does not provide any additional styles for this plugin.

To set up a form to use Abide, add the `data-abide` attribute to the `form` element:

```
<form data-abide> <!-- form elements --> </form>
```

Then for each form input element, you can add a `required` attribute and a `pattern` attribute with a value or an error message. Here is an example:

```
<label>Enter your surname:</label>
<input type="text" required pattern="^[a-zA-Z]+$">
<small class="error">Surname must be alpha characters</small>
```

In the previous code, the form is intended to accept only upper and lowercase alphabetic characters. The input field is marked as `required`, and an error message is displayed if something is incorrect.

In addition to being able to hard code a pattern, Foundation has several built-in patterns that you can use. For example, there is an `alpha` pattern that is the same as the one we hard coded in the previous example. So you can replace the pattern specification with the name of a pattern like this:

```
<input type="text" required pattern="alpha">
```

Check the JavaScript file you have with your installation of Foundation to determine the patterns that are built in. If you are really keen and want to add your own patterns, you can do that without editing the JavaScript file by adding patterns as you load Abide.

Summary

Congratulations! If you've reached this point and have a basic understanding of what each plugin does and how to set it up, you've done well. If you actually went through the setup and made each one work, you deserve an A+.

You can do amazing things with these plugins just by working with markup and styles. It is rare that you have to do anything more with JavaScript than installing it. But it does add a layer of complexity that makes it more of a challenge to set up and tweak plugins.

Good news is that you can now use Alerts, Tooltips, and Dropdowns to highlight things on your page. You can organize your content or navigation using the Section plugin. For advanced navigation, you have the Top Bar. You can use Magellan or Joyride to navigate within a page. Orbit, Clearing, and Reveal are all powerful plugins to enhance your presentation of specific content on a page. With the new Interchange plugin, you can optimize image loading for specific devices. With the Forms and Abide plugins, you can build professional forms.

In the next chapter, we'll introduce a totally different approach to install and customize Foundation. While a complete discussion of SASS with Foundation could be the topic of an entire book, we'll make sure you get enough to get started.

4
Advance with SASS

Up to this point in this book, we've assumed you were using a regular or custom download of Foundation from Zurb's website. We showed how to exploit the grid system and how to configure and customize each of Foundation's CSS components and JavaScript plugins. Now we'll help you advance with Foundation's SASS. After a brief look at what SASS is about, we'll show how to install Foundation with SASS.

When you download Foundation from the website, you could select the components to include. We'll show you how to do that after you have installed Foundation with SASS. Then we'll show how to customize your CSS with SASS variables, how to work with SASS functions and mixins, and how to extend classes using SASS. Each of these allows you greater flexibility and control of your style sheets.

In the last section of this chapter, we'll introduce you to designing with SASS. That's where you can take your HTML markup and quickly apply Foundation's grid mixins to style it into the layout you desire.

Understanding Foundation's SASS

In a nutshell, **SASS**, which stands for **Syntactically Awesome Style Sheets**, is a programming language that helps you structure and simplify your style sheets. The variation of SASS that Foundation uses is **SCSS**, which stands for **Sassy CSS**.

Everything you can do in CSS, you can do in SCSS, and a whole lot more. Start with variables. Remember how hard it is to set default colors for your website and use those colors in all the appropriate places? Define a variable and set the color there. Use that variable in your SCSS instead of hardcoding a specific color. Now when you want to change the color, you only change it once and your style sheets are updated with the new color in every place they're needed.

Toss in some functions and mixins. Now you can perform calculations and specify the common properties that can be reused. Mixins can accept parameters and return a different result based on those.

With SASS you can also extend and adapt the existing styles. This is useful when you want one style to be like another, except that you want to add or override some properties. Change the underlying style and your new style updates accordingly.

This is a brief overview of SASS and SCSS. For a more detailed explanation as you go through this chapter, visit the main website `http://sass-lang.com/` or search for keywords like SASS and SCSS development.

Installing Foundation with SASS

Earlier we showed how to download and install Foundation from the Zurb website. Now we'll see how to install and customize a SASS version of Foundation.

There are two approaches to this. The first approach is to download and compile a SASS version of Foundation from GitHub. This closely resembles what you get when you download a CSS version from the Zurb website. The second approach is to get a SASS version of Foundation by installing it as a Ruby Gem.

We'll cover the first approach here—download and install. Not only will it look more familiar, but it will also be easier to understand where the bits are and it will be more straightforward to update when a new version of Foundation becomes available. As you become more advanced, you can explore the second approach, the gem install, on your own.

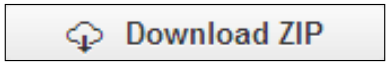
1. **Install Ruby and Rubygems**: For both methods you'll need a Ruby environment installed and working. Getting the Ruby environment up and running is beyond the scope of this book. The process is different for each operating system: Windows, OSX, and Linux. If it isn't already installed in your work environment, search for "ruby installation" and you'll quickly find links to download files and instructions. The main website for Ruby is `http://rubyonrails.org/`. Be sure to install RubyGems as you'll need that too.

   Once you have Ruby and RubyGems running, the steps are simple. You don't need to learn Ruby; you just need to be able to run Ruby commands from your terminal.

2. **Download the SASS version of Foundation**: When you first installed Foundation, you went to the download page at `http://foundation.zurb.com/download.php` and obtained the full or a customized version. The third installation option on that download page is **Install Foundation SCSS**. At present there is a button that looks like the following screenshot:

    1. Click on that button. Part way down the resulting page is a section titled **Using Sass Standalone**. In that section is a button that takes you to Foundation at GitHub. At present the button looks like the following screenshot:

    2. Click on that button. Or if you want to skip the Foundation download pages, you can go directly to GitHub, currently at `https://github.com/zurb/foundation/tree/scss-standalone`.
    3. In the lower-right corner of the GitHub page there is a button that says **Download ZIP**. At present it looks like the following screenshot:



    4. Click on that button to save the ZIP file to your computer. Unzip it and copy it to where you want your website. This is equivalent to the Foundation folder that you copied from the CSS download page, except that it contains the bits needed to customize Foundation in two subfolders, `scss` and `js`.

3. **Install the SASS version of Foundation**: Install this foundation folder into your website in exactly the same way you did for the CSS version downloaded from the Zurb website. Whether you made the foundation folder the root directory of your website, or put the foundation folder in as a subfolder of your root directory, do the same with the SCSS version.

4. **Install the SASS compiler**: Open your Terminal program (run `cmd.exe` in Windows). This is where you'll need Ruby and RubyGems. The first thing we'll do is to install the SASS compiler using the following command:

   ```
   gem install sass
   ```

5. **Compile your style sheets**: Navigate to the folder where you just installed Foundation and create a folder called `css` as a sibling to the `js` and `scss` folders that are already there. Now compile the SCSS files into CSS files:

   ```
   sass --update scss:css
   ```

   Congratulations! You've just generated your first Foundation style sheet. Look in your newly created `css` folder and you should see two files, `foundation.css` and `normalize.css`.

6. **Verify your installation**: To further verify that you've succeeded, take a copy of the `index.html` file from a CSS copy of Foundation that you downloaded directly from the Zurb download page. Put that `index.html` file in the foundation folder where you've just been working. Load that file in your browser and you should see exactly the same page you see when you load it from its original location.

Customizing Foundation with SASS

When you first installed Foundation, we recommended that you install a custom style sheet so you can tweak and override Foundation's default styles. You may still do that here. But the purpose of the SCSS version of Foundation is to let you generate your own CSS from the toolkit and add any customizations you want to make.

You can upgrade the SASS version of Foundation at a later date. The principle for modifying files is the same as for the CSS version. Create copies of the files you need to edit and leave all the Foundation files intact. There are only two files that you need to edit in the SASS version, the main SCSS file and the variables SCSS file. The setup is as follows:

1. In the `scss` folder make a copy of `foundation.css`. Call it `custom-foundation.css`.
2. In the `scss/foundation` folder, make a copy of `_variables.scss`. Call it `_custom-variables.scss`.
3. Edit the `custom-foundation.scss` file and change the line that imports the variables file to reference your new variables file. It should be as follows:

   ```
   @import "foundation/custom-variables";
   ```

4. Run the `sass` compiler again from the `foundation` folder to generate new CSS files. Look in the `css` folder again and you'll see another file called `custom-foundation.css`.
5. To verify that all these steps worked, edit the `index.html` file that you copied over earlier and link to `custom-foundation.css` instead of `foundation.css`. Load the `index.html` file in your browser and it should look exactly the same as before. From now on you'll use `custom-foundation.css` in all your web pages.

When you are ready to upgrade to a newer version of Foundation, go through the process again to download, unzip, and copy the new version on top of the existing one. Whatever you've done in your custom files will not be affected.

This is a bit simplistic. Surely you will make a copy and test the upgrade before overwriting a production website. The key point is that this process is very similar to downloading and installing or upgrading a CSS version of Foundation. But you get the advantage of being able to generate your own custom installation.

There is one more thing. We'll soon be editing SCSS files. You can recompile the SCSS files into CSS at any time by running the `sass --update scss:css` command again. Or if you prefer to have your SCSS compiled automatically each time you make a change, useful during development, you can run the `sass` command with the `--watch` option as follows:

```
sass --watch scss:css
```

Now your SCSS folder will be monitored and any time there is a change, `sass` will recompile automatically. You can stop this process by pressing *CTRL + C*.

Choosing Foundation components

The first thing we'll look at when customizing Foundation through SASS is choosing components and plugins. Here are the steps to accomplish this:

1. **Set up to enable and disable components**: Open the `custom-foundation.scss` file again in your favorite editor. There are two import statements. The first one imports your custom variables; we'll cover that later. The second one imports all the individual components. Comment out all the ones you don't want and you'll be left with the ones you do want.

   Alternatively, our preferred approach is to comment out all the component lines and then uncomment only the ones we want.

2. **Close the import statement**: Foundation uses one import statement to import all the component files. Filenames are separated by a comma and there is a semicolon after the last one. This means you have to make sure the last file in the list has a semicolon rather than a comma at the end. That quickly becomes a nuisance because it is easy to forget. Here is a way to make it easier to quickly enable and disable each component:
    1. In the `scss` folder create a file called `_custom.scss`. Put a comment in it to remind yourself what it's about when you open it at some later date. In SCSS, the comment syntax is the same as in CSS files. That is a `//` at the beginning of a single line comment. For a multiline comment open with a `/*` and close with a `*/`.
    2. At the end of the last component line in the list, change the semicolon to a comma. Then add a line below that with the file name `custom` in quotes and follow it with a semicolon. Now you can enable and disable components without having to make sure the last line ends with a semicolon.

    We suggest the name `custom` for a reason. We'll put the custom SCSS code in that file. When it compiles, the resulting CSS will be at the end of the style sheet. That way we can override Foundation styles using SCSS in the same way as we would with CSS, by loading a custom style sheet after our main style sheet.

3. **Enable the first two components**: To get started, disable all the components except the first two, `global` and `grid`. Then go back to your terminal and recompile the CSS files. Open the `index.html` file in your browser. This time you'll see that the content is there and is laid out in a grid, but the custom presentation, buttons, and so on are gone. It looks plain. That means you have successfully compiled a basic style sheet with only the global styles and grid classes.

    When editing and compiling SCSS files it only affects the CSS files. It does not affect JavaScript files. You'll still need to manage those separately to load only what you want. See the JavaScript installation section in *Chapter 3*, *Pick and Choose JavaScript Plugins*, for details on how to do this.

Customizing with variables

If you installed a custom download of Foundation from the Zurb website, not only could you selectively include components, but you could also customize several variables before downloading. With SASS you can customize all those variables and a whole lot more, any time you want. It is worth learning how to install and compile Foundation with SASS for this capability alone.

Open the custom variables file that you created earlier. It's in the `scss/foundation` folder. We suggested calling it `_custom-variables.scss`. Foundation has helpfully put a copy of the variables you can modify in this file. If a variable is commented out, its default value will be used.

Variables are defined in the component SCSS files that you'll find in the `scss/foundation/components/foundation` folder. When you uncomment and change a value in your custom variables file, you override the default and your value takes effect.

Let's do it. Search through the custom variables file and locate Grid Variables. Set the row width to `960`, the column gutter to `24`, and the total columns to `24`. Compile your SCSS files if it didn't happen automatically. Load the `index.html` file again. If the rightmost column in each row has moved to the right and there is a large space between that column and the other columns in that row, you've succeeded.

You can use your browser's developer plugin to confirm that the outer row width is now 960 px instead of 1000 px and the gutter between columns is 24 px instead of 30 px. But look at what else you've done. You now have a 24-cell grid! That's something you couldn't do when downloading a custom installation from the website.

Open the custom foundation CSS file and prove to yourself that you have a 24-cell grid. The `small-n` and `large-n` classes go all the way to 24.

Let's tweak a few more things, so you can get a little more comfortable with changing variables. First, set the number of cells in a row back to `12` and enable the `button` and `type` components by uncommenting those lines in `_custom-foundation.scss`. When you've done this, recompile your CSS and then reload the `index.html` file. It should be looking normal again.

When changing variables, it is recommended that you make a copy of the variable that you want to change. Leave the original intact for documentation purposes. Uncomment and change the value of the copy. When you come back to the variables file after a few months, you'll easily see what the original values were before you made changes. You can also find the default values by searching through the individual component's SCSS files; however, it is easier to simply retain them in the variables file.

Now make the following changes to the variables SCSS file:

- Search for where the body background color is defined. It is called `$body-bg`. Change that to `#ebebeb`.
- Look for where the colors are defined. Change the value of `$alert-color` to `#ebeb00`.

- Change the font family for headers. Search for the `$header-font-family` variable. Set its value to `"Times New Roman",Georgia,Serif;`.
- How about our small device breakpoint? The default is `768px`. Let's set it to `600px`. Look for the `$small-screen` variable and set it to `600px`.

That's enough to see the power of these variables. Save your changes and recompile. Then load up the `index.html` file again.

The first thing you'll notice is that the background color has changed. It was white, now it's gray. That's not a big deal. It's easy to set a background color on the `<body>` tag through CSS, so we didn't really save much effort. But let's also look at the other changes we just made.

The **alert** button on the page is now a shade of yellow. That means the buttons component picked up on the new color. Hover over the **alert** button. The hover style has also changed. If you were to add the `disabled` class to the button, you would see that it has also changed to a variation of the yellow color. Any other components that had been installed and used the alert color would also have changed it. By changing one variable, we've changed the alert color across the website. That's powerful. If you've ever tried to maintain consistent colors and color variations through CSS, you'll really appreciate this.

How about the font family for headings? Regardless of the size, they have changed to a `serif` font from `sans-serif`. And once again, any component that uses any variation of heading tags will be affected. Changing one variable has a big impact.

Narrow your browser and see where the breakpoint is. As it narrows from 768 px to 767 px, nothing happens. It's only when you drop below 600 px that the columns in the rows stack. Foundation has looked after all the places in your style sheet where that needs to be changed. You don't have to search through the style sheet to find all those places. You simply set one variable and recompile. The job's done.

As you scroll through the custom variables file, you'll notice that the general variables, the ones that affect many components, are at the top. Below those are variables that affect only one component. But the same principle applies. If you want to use that component, it is easier to change one variable than to search a style sheet to find all the places where a color needs to change.

Look at the color styles for the Top Bar plugin. There are still many variables to consider. But it is much easier to understand and change your Top Bar to another color by setting those SCSS variables than it is to do so by changing Top Bar's CSS styles.

Let's create our own styles with some variables. Foundation uses color classes in components. When it does that, it always combines a color class (for example, primary color) with another class (for example, button) to display an object in that color when both the classes are used. But it doesn't define the color classes on their own, so you can use them anywhere.

The following code demonstrates the use of variables to set colors:

```
.primary-color { color: $primary-color; }
.secondary-color { color: $secondary-color; }
.alert-color { color: $alert-color; }
.success-color { color: $success-color; }
.warning-color { color: $warning-color; }
```

It's quite simple, really. We just used the variable name in place of a color. Now we have stand-alone classes for each of our colors. Oh, we also introduced a new color variable, `$warning-color`, because we thought there should be something between the `success` and `alert` colors. When you add this code to your custom SCSS file, make sure you add that variable first, as follows:

```
$warning-color: #ffff33 !default;
```

By adding the `!default` keyword to the variable definition, we are able to override its value elsewhere. The natural place to do that is in the custom variables file.

Understanding functions and mixins

Now that we understand variables, let's move on to functions and mixins. A function is SCSS code that will perform some operation(s) and return a single value. A mixin is used to combine styles that you want to use repeatedly. By including a mixin you get all those styles automatically without having to remember and repeat them.

There are some functions defined near the top of your custom variables file. Let's look at one of those:

```
@function emCalc($pxWidth) {
  @return $pxWidth / $em-base * 1em;
}
```

This function converts a number representing pixels into em units. It is a simple function that allows you to think in pixels, yet specify your CSS in em units. The following is an example:

```
.emphasis {
  font-size: emCalc(24);
}
```

This results in the following CSS:

```
.emphasis {
  font-size: 1.5em; }
```

The function `emCalc(24)` returned the value `1.5em`. To call a function you simply use the function in place of a value or variable.

The following is a simple mixin that you can reuse to center a container:

```
@mixin center() {
  position: relative;
  margin-left: auto;
  margin-right: auto;
  float: none !important;
}
```

You've probably seen this code before as it is a common way to center a container that doesn't center with the `text-align` property. By building it into a mixin, you can include it elsewhere in your SCSS. For example:

```
div.standout {
  background: yellow;
  width: 50%;
  @include center;
}
```

We'll use this `standout` class when we want a `div` element to stand out on the page. It will have a yellow background, take up 50 percent of its container, and be centered within that container. The complete CSS, when this gets compiled, looks as follows:

```
div.standout {
  background: red;
  width: 50%;
  position: relative;
  margin-left: auto;
  margin-right: auto;
  float: none !important; }
```

Everything in the `center` mixin was added to the `div.standout` class. We didn't have to remember all the properties needed to center the container. If for any reason we wanted to add something to the `center` mixin, we could do that and recompile; it would update all the individual styles that included that mixin.

With a function we simply reference it by name; however, with a mixin we need the `@include` directive followed by the name of the mixin. Both functions and mixins can accept arguments.

Let's use one of Foundation's built-in mixins to generate custom styles. Assume we are working on an old website where we can't change the classes that are used, but we can change the CSS that defines those classes. There is a `main-container` class that we would like to take on the properties of an outside row. The code is very simple and is as follows:

```
.main-container {
  @include grid-row();
}
```

Compile that and look at your style sheet. The styles for `main-container` will be exactly the same as for the `row` class as follows:

```
.main-container {
  width: 100%;
  margin-left: auto;
  margin-right: auto;
  margin-top: 0;
  margin-bottom: 0;
  max-width: 60em;
  *zoom: 1; }
  .main-container:before, .main-container:after {
  content: " ";
  display: table; }
.main-container:after {
  clear: both; }
```

With this simple code you've taken one step towards back porting Foundation into an existing website, that perhaps wasn't even a responsive site when you started. The preceding code was equivalent to a simple outside row, which may be sufficient for redefining the `main-container` class. However, there may be times when you want to be more sophisticated. That's when we'll extend classes.

Foundation has many functions and mixins built in. Scan through the SCSS files and you'll find plenty to draw on. There are other sources of SCSS functions and mixins. If you use Compass instead of Sass to compile your SCSS files, you have ready access to a whole library of useful functions and mixins.

Extending classes

An alternative to including a mixin with predefined properties is to extend classes. It defines a new class that inherits all the properties of another class. The following is an example:

```
.emphasis {
  font-weight: bold;
  background-color: #ffffaa;
  font-size: emCalc(24);
}
.special-emphasis {
  @extend .emphasis;
  color: blue;
}
```

The preceding SCSS code compiles into the following CSS:

```
.emphasis, .special-emphasis {
  font-weight: bold;
  background-color: #ffffaa;
  font-size: 1.5em; }

.special-emphasis {
  color: blue; }
```

Now we can use `special-emphasis` and it will have all the characteristics of `emphasis` and the additional property of blue text.

One thing that's beautiful about the `@extend` directive is that it extends all the occurrences of a class. When we included the `grid-row()` mixin in the `main-container` class, that only made the `main-container` class equivalent to the `row` class by itself. There is another class in our website that needs to be equivalent to an inner row. By using `@extend` and making both the classes inherit all the characteristics of the `row` class, in all occurrences, we get what we need:

```
.main-container,
.feature-box {
  @extend .row;
}
```

Now look through your compiled style sheet. We won't copy the result here because it is quite lengthy. But every time you see the `row` class used, you can instead use the `main-container` or `feature-box` classes. They are all equivalent.

In our website `main-container` was the outer container. It is equivalent to an outer row in Foundation. Further into the website, there is the `feature-box` container. A `feature-box` class within a `main-container` class is identical to a row within a row, what we call an inner row. These two combinations have identical properties as shown in the following code:

```
.row .row { properties }
.main-container .feature-box { properties }
```

Inside our `feature-box` container we have four equal-width containers. That's the same as four three-cell columns in Foundation. So we'll use the `@extend` directive again to make those equivalent. In this case there is only one class called `feature-box-inner`. Here is how we make that equivalent to the `large-3` and `column` combination:

```
.feature-box-inner {
  @extend .large-3;
  @extend .column;
}
```

These are simple lines of SCSS code. When you look at your compiled style sheet, you'll see that the `feature-box-inner` class is the same as both the `large-3` and `column` class. Wherever the `column` class is specified, the `feature-box-inner` class is also specified. Now we can use the `feature-box-inner` class in place of both the `large-3` and the `column` classes together.

When we remove the original style sheet from this website and link our compiled Foundation style sheet, it works exactly as we wanted.

It isn't always feasible to combine two or more classes into one. When two classes have conflicting properties, the second one to be extended will win. In our example, `large-3` and `column` are designed to work together, so it is safe to combine them into one.

In practical terms this is only a small part of the overall website. To do a complete conversion, we have to look at many other parts on the page and add a lot of refinements to get everything to look decent. But with these examples you can see the power of extending Foundations classes to create your own variations and combinations.

Designing with SASS

Where do you start when designing your website layout? You probably start with your HTML markup and add Foundation's grid classes until you have something that meets your requirement. You'll be happy to learn that you can make this process even simpler by using Foundation's SASS grid components.

Let's start with a simple three-part layout:

```
<header>
<main>
<footer>
```

We want each of these to be an outside container, an outer row. So we'll add the SCSS code to do that. Put the following code in your custom SCSS file:

```
header { @include grid-row(); }
main { @include grid-row(); }
footer { @include grid-row(); }
```

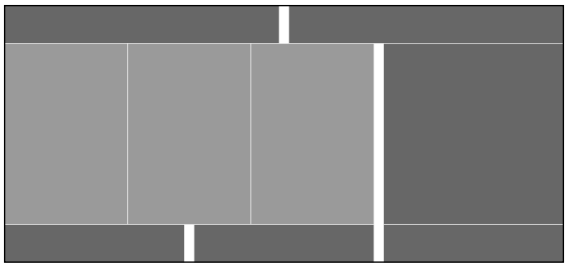
When you compile your SCSS, you will see that the header, main, and footer elements all have the same properties that an outer row has in Foundation's style sheet. The beauty of this is that you do your markup in HTML as usual, mirror that in SCSS, and add appropriate mixins.

To show they are equivalent, here are the properties associated with the `.row` class and the `footer` tag:

```
.row {                          footer {
  width: 100%;                    width: 100%;
  margin-left: auto;              margin-left: auto;
  margin-right: auto;             margin-right: auto;
  margin-top: 0;                  margin-top: 0;
  margin-bottom: 0;               margin-bottom: 0;
  max-width: 60em;                max-width: 60em;
  *zoom: 1; }                     *zoom: 1; }
  .row:before,                    footer:before,
  .row:after {                    footer:after {
    content: " ";                   content: " ";
    display: table; }               display: table; }
  .row:after {                    footer:after {
    clear: both; } }                clear: both; } }
```

As you can see, the properties are identical. In actual fact there are more properties associated with the `.row` class, but only when used in conjunction with another class like `collapse`. But since we only need the footer to be an outside row, this is all we need.

In *Chapter 1*, *Get the Most from the Grid System*, we used a blog-type website as an example. Let's do that again, only this time we'll add the layout with SCSS mixins. This is what we are aiming for:



There is a header, main, and footer section. The header has two equal parts. The footer has three. The main section has a three-column content area and a sidebar.

Let's expand our HTML markup and our corresponding SCSS to implement this. Here is the markup:

```
<header>
  <div>Header Left</div>
  <div>Header Right</div>
</header>
<main>
  <aside>Sidebar</aside>
  <section>
    <article>
      <div>Content Left</div>
      <div>Content Center</div>
      <div>Content Right</div>
    </article>
  </section>
</main>
<footer>
  <div>Footer Left</div>
  <div>Footer Center</div>
  <div>Footer Right</div>
</footer>
```

This HTML markup is easy to follow and easy to construct. For the most part it accurately reflects the structure that we want. The one exception is that we put the sidebar ahead of the content area in the main section. That's because we construct the HTML in the order we want it to appear on small devices. We'll adjust to change the order on regular displays.

We have added some text in each of the individual content areas so that there is some initial content on our page. Wherever we didn't add text, the tag is there for structural purposes. Having this text will make it easier to confirm that we have the layout we want.

Here is the SCSS code to implement this blog layout for regular displays:

```
header { @include grid-row();
  div { @include grid-column(6); }
}
main { @include grid-row();
  section { @include grid-column(9, $pull:3);
    article { @include grid-row(nest);
      div { @include grid-column(4); }
    }
  }
  aside { @include grid-column(3, $push:9);}
}
footer { @include grid-row();
  div { @include grid-column(4); }
}
```

As you can see, the preceding SCSS code is structured exactly the way the HTML markup is structured. That's the beauty of it.

Compile this and then load the HTML in your browser. You'll see that the text we entered there is displayed in a grid layout, positioned as we specified.

We did take a few shortcuts. Each time there was a row with equal columns, we only needed to include the SCSS mixin once. That affected the two columns in the header, the three columns in the content area, and the three columns in the footer. Had any of these been different from their siblings, we would have had to identify them separately.

In the first example, we introduced the `grid-row()` mixin. In this example we introduced the `grid-column()` mixin and used attributes in each of these when it was appropriate.

Look at the `section` and `aside` tags. We added the `$push` and `$pull` attributes to push the sidebar to the right and pull the content area to the left. Look at the `article` tag. We added the `nest` value to the inner row definition to identify it as an inner row. That means the `article` tag will have the same properties as the two `.row` classes together in the style sheet.

There is nothing sacred about applying our SCSS mixins to named tags. We could just as easily have made everything a `<div>` element with an appropriate class. Using the HTML tags the way we have means they are context sensitive. If you look at the compiled style sheet, you'll see that `article` only exists in the context of the `main section article` classes together. Its parent classes are included.

For development purposes, we have a mixin that we include on content areas when we are designing a layout. It helps provide a better visual result and makes it easier to show clients what they can expect in the final product. The following is that mixin:

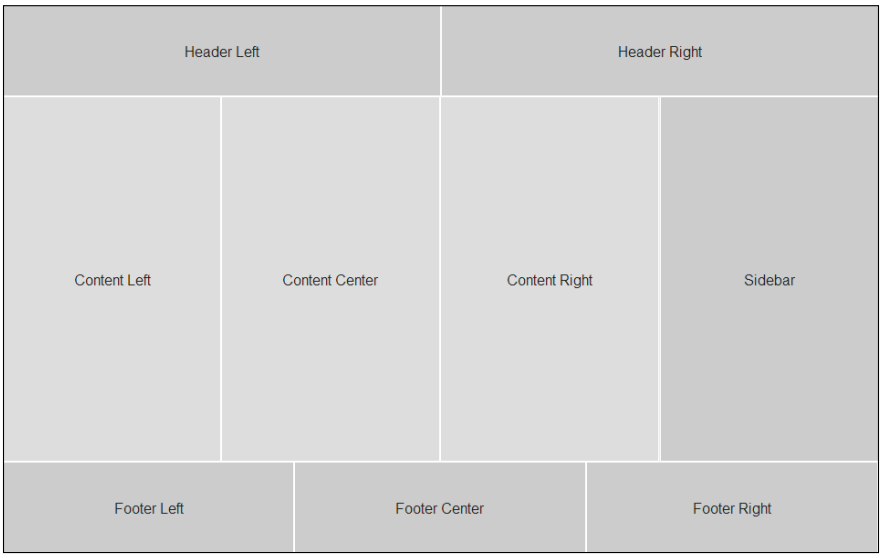
```
@mixin layout($height:100px, $bg-color:#ccc) {
  background: $bg-color;
  border: 1px solid white;
  height: $height;
  text-align: center;
  padding-top: $height/2 - $em-base/2;
}
```

This mixin adds some styles to each content area. And it has a couple of attributes, height and background color, that you can set when you use it.

The following is our blog example with this code included in each of the content areas:

```
header { @include grid-row();
  div { @include grid-column(6);
  @include layout(); }
}
main { @include grid-row();
  section { @include grid-column(9, $pull:3);
    article { @include grid-row(nest);
      div { @include grid-column(4);
        @include layout(400px, #ddd); }
    }
  }
  aside { @include grid-column(3, $push:9);
    @include layout(400px); }
}
footer { @include grid-row();
  div { @include grid-column(4);
    @include layout(); }
}
```

Compile and load your page in your browser again. This is what you should see now:



It is our target website on regular displays. For the header and footer areas, we relied on the default values for the `@layout` mixin. For the main section we set the height to `400px`, and for the main content area we set the background to a lighter color. Use this mixin as it is or modify it to suit your preferences.

Media queries can be nestled into your SCSS code in appropriate places. Or you can wrap the entire layout in a media query, copy it, and modify it for another device. There is no right or wrong way.

When you copy and modify, it is easier to visualize the entire layout for each device. It works when you want to design for one device and then adapt for another. Embedding a media query works better when you want to think about more than one target device as you design your layout.

Now you can start designing with SASS, quickly mocking up a site, showing it to people for feedback, and adjusting accordingly.

Summary

In this chapter we've introduced you to SASS in Foundation. There is much more that we were not able to describe here. Every component and every plugin has a SASS file associated with it.

Install Foundation with SASS the way we've suggested, and it will look like a regular or custom download from the Zurb website. But now you'll have the added capability of customizing your layout and styles through SASS.

Change variables to affect a component. Include mixins when appropriate. Extend classes to make your own variations. Use SASS to quickly lay out your website. Now you have all these tools at your disposal.

Enjoy the journey!

Index

J

K

L

M

N

O

P

R

S

T

V

W

Z

Thank you for buying

Getting Started with Zurb Foundation 4

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

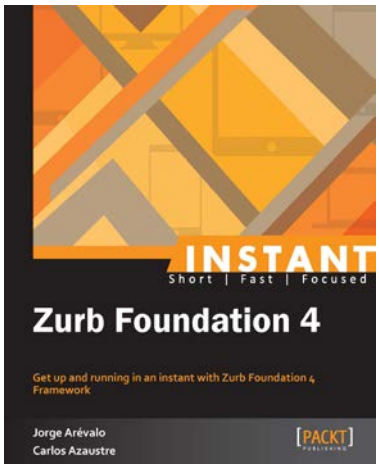
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING



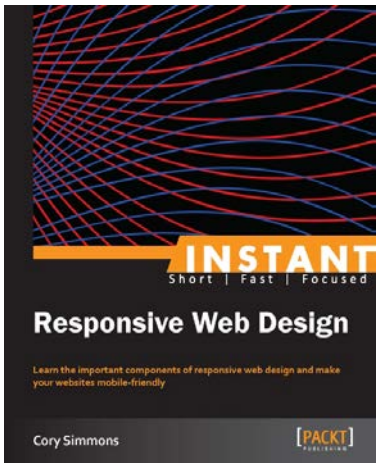
Instant Zurb Foundation 4 [Instant]

ISBN: 978-1-78216-402-9 Paperback: 56 pages

Get up and running in an instant with Zurb Foundation 4 Framework

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results
2. Construct responsive and mobile-ready web pages without worrying about browser-related issues. Just code once and it will be compatible with all browsers and display sizes
3. Learn to use Foundation 4 features with actual code examples and ample screenshots

---



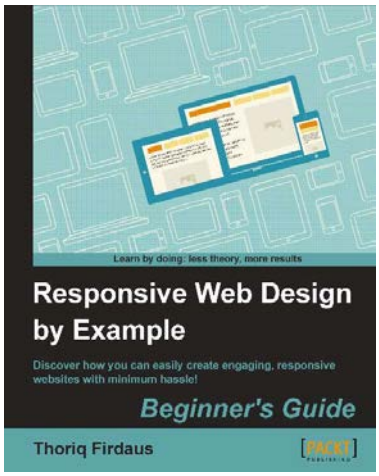
Instant Responsive Web Design [Instant]

ISBN: 978-1-84969-925-9 Paperback: 70 pages

Learn the important components of responsive web design and make your websites mobile-friendly

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results
2. Learn how to make your websites beautiful on any device
3. Understand the differences between various responsive philosophies

[PACKT] open source
community experience distilled
PUBLISHING



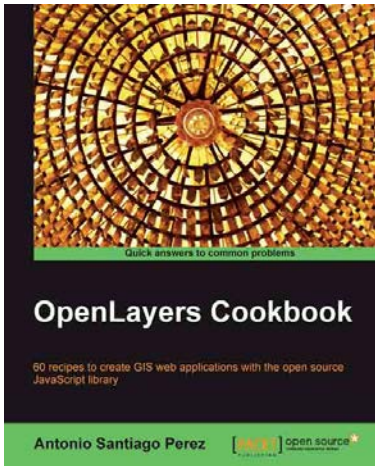
Responsive Web Design by Example Beginner's Guide

ISBN: 978-1-84969-542-8 Paperback: 338 pages

Discover how you can easily create engaging, responsive websites with minimum hassle!

1. Rapidly develop and prototype responsive websites by utilizing powerful open source frameworks
2. Focus less on the theory and more on results, with clear step-by-step instructions, previews, and examples to help you along the way
3. Learn how you can utilize three of the most powerful responsive frameworks available today: Bootstrap, Skeleton, and Zurb Foundation

---



OpenLayers Cookbook

ISBN: 978-1-8495-1-784-3 Paperback: 300 pages

60 recipes to create GIS web applications with the open source JavaScript library

1. Understand the main concepts about maps, layers, controls, protocols, events, and so on
2. Learn about the important tile providers and WMS servers
3. Packed with code examples and screenshots for practical and easy learning