Bleeding Edge Press

CREATING INTERFACES WITH BULMA

Jeremy Thomas, *creator of Bulma*
Oleksii Potiekhin, Mikko Lauhakari,
Aslam Shah & Dave Berning

Creating Interfaces with Bulma

By Jeremy Thomas, creator of Bulma, Oleksii Potiekhin, Mikko Lauhakari, Aslam Shah, and Dave Berning

Creating Interfaces with Bulma

ISBN **9781939902498**
Published by: Bleeding Edge Press, Santa Rosa, CA 95404
Title: Creating Interfaces with Bulma
Authors: Jeremy Thomas, Oleksii Potiekhin, Mikko Lauhakari, Aslam Shah, & Dave Berning
Acquisitions Editor: Christina Rudloff
Editors: Troy Mott & Dave Berning
Website: bleedingedgepress.com

Table of Contents

Foreword by Jeremy Thomas

I discovered CSS almost by accident in 2007. During an accessibility class, the teacher was emphasizing the need to separate content from styling, and told us it could be achieved with CSS. It was a breakthrough for me: no need for Dreamweaver and complex table layouts anymore. I could write in a simple language that would translate my rules into visual interactive interfaces. I didn't know this event would eventually define the start of my career as a web developer.

For the next 10 years, while I would teach myself various web development tools (PHP, JavaScript, Ruby, Node...), CSS would remain my strongest skill, and the reason why clients and companies would hire me. In the meantime, new CSS features were being developed and adopted by browsers. I was already pretty happy with shadows, rounded corners, custom fonts, and gradients, since they didn't require PNG hacks or convoluted workarounds anymore. But by the end of 2015, a new layout model was becoming increasingly popular. It was called Flexbox.

Flexbox was a game changer: instead of relying on floats, clears and a complex markup to define columns, you could define a Flexbox container with automatically resized columns and you had yourself a grid system! This solution would also drastically simplify the HTML markup. I knew Flexbox could be used to develop something new, something powerful, something exciting! But I didn't know what exactly.

By the time I'd discovered Flexbox, I was already using a small Sass framework I'd built and maintained myself over several months. I used it to kickstart various CSS projects, both personal and professional. Flexbox turned out to be the missing piece: the main appeal of a CSS framework is to simplify the process of defining page layouts, and Flexbox's syntax was the perfect candidate for a clearer and more flexible markup. While Bulma was initially a CSS generator I was working on that was making use of "capsules" (hence the name) as modular components, I decided to ditch the idea completely and rather combine my Sass framework with my recent Flexbox knowledge into a new modern CSS framework. Bulma was born.

Since I've always been an open source advocate, I decided to post my small framework on GitHub and share it across various tech and social websites. I thought "If this small framework I built solves a problem of mine, there's a chance it might solve a problem for someone else too." While the initial launch was really quiet, it suddenly went viral. Bulma was trending on GitHub, reached the Hacker News and Product Hunt homepages, and was shared hundreds of times via Twitter. I realized I had built something not only interesting,

but actually useful. I remained cautious though. Maybe Bulma's popularity was only a sudden burst of excitement that would fade away soon. But it did not.

Two years into the project, Bulma has been starred 24,000 times on GitHub, and downloaded or installed more than 1 million times. 150 contributors have helped close 860 issues and merge almost 300 pull requests. It shows how the open source community can turn a small CSS project into a major asset for web developers. And considering how it spawned gorgeous websites and made lots of businesses thrive, there is no question that Bulma will continuously grow and remain a widely used tool in the future.

I've acquired a lot of knowledge in the process, whether it's new CSS techniques or better writing skills. I've also seen many fans express their love for Bulma, praising its simplicity and ease of use. But I think the best reward for me is to know that I've been able to help thousands of people make the web a place of their own.

Preface

Who is this book for?

This book is for any designer or developer willing to understand how to use Bulma, and learn how to use Bulma's components and layout system to create their own web interface.

Even if you are not already familiar with Bulma, it only takes a few minutes to get acquainted with the framework.

What do you need to know prior to reading?

You don't need to know Bulma to read this book! You only need to have an understanding of how HTML and CSS work, but you don't need an in-depth knowledge since Bulma's purpose is to avoid writing CSS!

You also need a **code editor**: Sublime Text, Atom, Notepad++, IntelliJ, Vim, Emacs, etc. The only requirement is for your editor to have syntax highlighting and to be able to save a file with a specific extension (like .html or .css).

You will also need a modern browser: Google Chrome, Mozilla Firefox, Microsoft Edge or Apple Safari.

The online book publisher example

All of the code for the sample project in this book can be found at:

https://github.com/troymott/bulma-book-code

What will this book provide?

This book is a **step-by-step guide** that will teach you how to build a web interface from scratch using Bulma.

The example website that you will build is an administration interface for an online book publisher, where users can log in to manage three content types: Books, Customers, and Orders. This interface has been chosen because it satisfies all of the requirements for

common CRUD (Create/Read/Update/Delete) functionalities, which exist in any type of website or CMS. You can access all of the code for this example on **Github** (https://github.com/troymott/bulma-book-code).

By the end of this book, you will understand how to:

- Create layouts with Bulma
- Work with components in Bulma
- Design specific elements for your UI
- Extend components with your own setup

The book will also show you how Bulma can be integrated with JavaScript through the following frameworks: React, Angular, VueJS, and Vanilla JS.

Author bios

Jeremy Thomas has been a web designer for more than 10 years. While studying graphic design in France, he discovered CSS during an accessibility class and instantly fell in love with the language. That's when he decided to make a career out of it. He has worked with eCommerce companies, agencies (Sony, Microsoft, Louis Vuitton, freelancing, tech start-ups, code teaching).

By the beginning of 2016, Jeremy had developed a small framework that he was using himself for kickstarting his projects, and decided to share it for free to the world: Bulma was born. Still active in the open source community, he has launched other useful web resources like MarkSheet, CSS Reference, HTML Reference and Web Design in 4 minutes. His goal is to continuously share the knowledge he acquires through his daily work.

Book co-authors and contributors

Oleksii Potiekhin is a web developer by profession and by destiny with more then nine years of production experience in developing and designing GUIs on different platforms and technologies. He has worked with: Volvo, Scania, Volkswagen, Renault, John Lewis Partnership, Thomson Reuters, etc. He fell in love with Bulma in 2017 because it provides everything you need to build a modern UI for any kind of project.

Mikko Lauhakari is a developing web-creative, or just simply a web nerd. He's had a passion for the web since the last bubble burst. With a background in web programming studies at Kalmar University, Sweden, he has a wide knowledge base of different programming languages.

Aslam Shah is a Senior JavaScript Developer at Risk.Ident GmbH. He has 5+ years of experience in developing front-end interfaces for small to large-sized companies and believes that technology never stops evolving, and that we have to learn new things every single day to keep ourselves up to date; we shouldn't be scared of moving from old things to new ones.

Dave Berning has been a front-end web developer for more than six years. He graduated from the University of Cincinnati where he learned to create interactive websites with HTML, CSS, and JavaScript. David builds rich progressive web applications with Vue and React. He is also a writer for Alligator.io, and organizer of the CodePen Cincinnati meetups where he leads workshops and discussions about the latest technology in the field. You can find him almost anywhere on the internet as **@daveberning**.

Technical Reviewers

We would like to the thank the following technical reviewers for their early feedback and generous, careful critiques: Ivan Ković, François-Xavier Costanzo, Dario Castañé, Stanley Eosakul, Samantha Baita, Aaron Ang, and Dave Berning.

1 Understanding Bulma, terminology, and concepts

If you're reading this book, there's probably a good chance that you've heard of Bulma. Bulma is a lightweight configurable CSS framework that's based on Flexbox. Flexbox is a relatively new CSS spec that has good browser support.

Bulma makes using Flexbox a breeze and handles all of the hard work of Flexbox for you, so you don't need to know any Flexbox to get started. However, knowledge of the CSS spec is preferred.

This chapter covers Bulma at a high level to get you familiar with Bulma, its terms, and its concepts.

How is Bulma unique?

Here are a few reasons why Bulma is different than other CSS frameworks:

- **Modern**: All of Bulma is based on CSS Flexbox.
- **100% responsive**: Bulma is designed to be both mobile and desktop friendly.
- **Easy to learn**: Most users get started within *minutes*.
- **Simple syntax**: Bulma makes sure to use the minimal HTML required, so your code is easy to read and write.
- **Customizable**: With over 300 SASS variables, you can apply your own branding to Bulma.
- **No JavaScript**: Because Bulma is CSS-only, it integrates gracefully with any JavaScript framework (Angular, VueJS, React, or just plain Vanilla JavaScript)

Simple columns system

Bulma is mostly famous for its straightforward columns architecture:

```
<div class="columns">
  <div class="column">
```

```
    <!-- First column -->
  </div>
  <div class="column">
    <!-- Second column -->
  </div>
</div>
```

That's it! It only takes two classes (`columns` for the container, and `column` for the child items) to have a set of responsive columns. You don't have to specify any dimensions: both columns automatically take 50% of the width.

If you want a third column, you can just add another column:

```
<div class="columns">
  <div class="column">
    <!-- First column -->
  </div>
  <div class="column">
    <!-- Second column -->
  </div>
  <div class="column">
    <!-- Third column -->
  </div>
</div>
```

Each column will now take up 33% of the width. No additional change is required. Continue this and add as many columns in as you want. Bulma will automatically adjust the size for you.

Readability

Bulma is easy to learn because it's easy to read. For example, a Bulma button simply uses the class name `button`.

```
<a class="button">
  Save changes
</a>
```

To extend this button, Bulma provides **modifier classes**. They exist only as a way to provide the base button with *alternative* styles. To make this button use the primary turquoise color and increase its size to large, just append the classes `is-primary` and `is-large`.

```
<a class="button is-primary is-large">
  Save changes
</a>
```

Tip: You might want to stick with the “primary”, “secondary” naming conventions. This will help give some meaning to your styles and it leaves it open for customization down the road.

Customizable

Bulma has more than *300 variables*, making almost any value in Bulma easy to override, allowing you to define a very personalized setup.

By using SASS, you can set your own initial variables, like overriding the blue color value, or the primary font family, or even the various responsive breakpoints.

```
// 1. Import the initial variables
@import "../sass/utilities/initial-variables"
@import "../sass/utilities/functions"

// 2. Set your own initial variables
// Update blue
$blue: #72d0eb

// Add pink and its invert

$pink: #ffb3b3
$pink-invert: #fff

// Add a serif family
$family-serif: "Merriweather", "Georgia", serif

// 3. Set the derived variables
// Use the new pink as the primary color
$primary: $pink
$primary-invert: $pink-invert

// Use the existing orange as the danger color
$danger: $orange

// Use the new serif family
$family-primary: $family-serif

// 4. Import the rest of Bulma
@import "../bulma"
```

Each Bulma component also comes with its own set of variables:

- `box` has its own shadow
- `columns` have their own gap
- `menu` has its own background and foreground colors

- `button` and `input` have colors for each of their states (hover, active, focus...)
- etc.

Each documentation page comes with the list of available variables to override.

Modular

Because Bulma is split into dozens of files, it's easy to only import the parts you actually need.

For example, some developers only want the columns. All they have to do is create a custom SASS file with the following code:

```
@import "bulma/sass/utilities/_all"
@import "bulma/sass/grid/columns"
```

This will only import the `columns` and `column` CSS classes.

Columns

Flexbox is a one-dimensional grid system, providing you with either rows or columns. In Bulma, you develop websites with columns in mind and wrap your columns inside a row or wrapper. Here is the most basic functionality of Bulma.

You start off with a `columns` row.

```
<div class="columns">

</div>
```

Inside of the `columns` row, you can add a single `column` or as many as you like. Bulma and Flexbox size your `column` depending on the number of columns added in a `columns` row.

```
<div class="columns">
  <div class="column">

  </div>
</div>
```

In this example, the `column` is 100% of the browser width, because there is only one column.

```
<div class="columns">
  <div class="column">
```

```
    </div>

    <div class="column">

    </div>
  </div>
```

Now, each `column` is not 50%. This was explained briefly in the introduction, but it's worth mentioning again. The more columns you add, the smaller they become. If you have three columns, each will be 33.33% wide, and with four columns, each column becomes 25% wide.

Modifiers

Modifiers are extra CSS classes that you add to your HTML in order to change its appearance. For example, let's look at a `<button>` and see how adding a modifier can change its appearance.

```
<button class="button">I'm a button</button>
```

So far its pretty generic, with not much going on. However, let's change it to a turquoise color that Bulma ships with. To change the color to a "primary" color of your theme, use the `is-primary` modifier.

```
<button class="button is-primary">I'm a button</button>
```

Now the button is turquoise. But let's not stop there. You can continue adding modifier classes to this button in order to change its appearance. Let's make it a "ghost" button or a hollow button with an outline.

```
<button class="button is-primary is-outlined">I'm a button</button>
```

You can also use the `is-loading` modifier class to show an animated loading GIF on your button. This shows the user that a process is going on, like when you submit a form.

Note: All modifiers in Bulma start with `is-` or `has-`.

It's considered best practice to leverage Bulma as much as possible before adding custom classes. If you overwrite the styles of something, continue using existing classes.

Components

Bulma ships with components, which are pre-styled chunks of code that serve a certain purpose. With components, you have to follow a specific HTML structure.

Reference Bulma's documentation for more information and examples of components.

Here is an example of a card component:

```
<div class="card">
  <header class="card-header">
    <!-- header content -->
  </header>

  <div class="card-content">
    <div class="card-image">
      <!-- card image -->
    </div>
  </div>

  <footer class="card-footer">
    <!-- footer content -->
  </footer>
</div>
```

Other components are: menu, dropdown, message, and modal.

Helper classes

Helper classes (a.k.a. utility classes) are modifiers that you can add to *help* structure your content and/or your user interface. These should not be confused as traditional modifiers that change the *look* of your component or element. These helper classes assist with user interface positioning.

Some examples of helper classes:

- `is-marginless`: Removes all margins.
- `is-unselectable`: Prevents the text from being selectable.
- `is-pulled-left`: Moves the element to the left.

There are other types of helpers, such as "responsive helpers" and "typography helpers" that assist with responsiveness and text respectively.

Summary

This chapter has introduced you to many Bulma concepts, but here are some further useful Bulma resources:

- **Bulma Documentation**
- **Bulma Blog**
- **Bulma Expo**

Next up, we'll examine how to create and control forms in Bulma.

Creating and controlling forms with Bulma

2

Let's dive right into creating user interfaces with Bulma. In this chapter, you create a *full screen* login form. This will give you a solid understanding of Bulma and give you the tools you need to start integrating Bulma. Some things to take away from this chapter are: the use of Bulma working with forms and *why* you leverage Bulma and *when*.

This login form that you'll create will contain two form inputs (one for the email and one for the password). This will be vertically and horizontally centered in a full screen `<div>`.

To see the full code of the example used in this book take a look at the **book's accompanying GitHub page**.

Template requirements

In order for the login page to work properly, you must follow the HTML5 web standard as well as the following tags:

```
<!DOCTYPE html>
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/font-awesome/
4.7.0/css/font-awesome.min.css">
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/bulma/
0.6.1/css/bulma.min.css">
```

All of these parts are combined in a **valid HTML5 template**:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Login</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/font-
awesome/4.7.0/css/font-awesome.min.css">
```

```
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/
bulma/0.6.1/css/bulma.min.css">
  </head>
  <body>
    <!-- The rest of your code will go here -->
  </body>
</html>
```

While this page is valid, it is not showing anything yet, so let's create our fill screen `<div>`.

Bulma provides a `hero` class: it creates a large imposing banner that is useful to showcase something in particular (in this case, the login form). This `hero` class comes with **modifiers** which, when combined with the base class, enable us to choose an *alternative* style for the "hero."

Inside the `<body>`, add the following snippet:

```
<section class="hero is-primary is-fullheight">
  <div class="hero-body">
    Login
  </div>
</section>
```

In addition to the `hero` class, leverage two Bulma modifiers: `is-primary` and `is-fullheight`. As stated above, modifiers "modify" the element that it's attached to. In this case, `is-primary` adds the default primary color (turquoise) and `is-fullheight` makes the `<section>` increase the height of the `<section>` to 100% of the browser's height.

You can now see that the whole viewport is turquoise, with "`Login`" written in white on the left side. The `hero-body` ensures that this text is **vertically centered**.

Tip: If you don't see a turquoise page, make sure you have included all of the different assets, and that you are connected to the internet.

Centered layout

Before implementing the login box, be sure to first set up the **layout**. You want the box to be both *horizontally* and *vertically* centered:

- `container`: Makes sure that the box will have a maximum width, and won't reach the edges of the page on wider viewports.
- `columns`: Is a wrapper for the single column.
- `column`: Will be *horizontally* centered.
- `box`: With its white background and shadow allows its content to be readable on this turquoise webpage.

```
<section class="hero is-primary is-fullheight">
  <div class="hero-body">
    <div class="container">
      <div class="columns is-centered">
        <div class="column">
          <form class="box">
            Login
```

```
          </form>
        </div>
      </div>
    </div>
  </div>
</section>
```

Even though you are using `is-centered`, the content doesn't look centered. It's because by default, each Bulma column is automatically resized to fill the horizontal space. Since you only have one column, it takes up **100%** of the width.

Tip: Try to add a second column, and notice how each column now takes up 50% of the horizontal space.

Since you don't want the login box to be too wide, resize this column.

Resizing the single column

You only need a single column, but you want that column to be **centered** and **responsive**. Luckily, Bulma provides modifiers that allow you to center columns, and specify a different column size for each breakpoint.

To achieve this, append the following modifiers to the form wrapper. Each one serves a specific purpose.

- `is-5-tablet`: Restricts the wrapper to be 5/12 columns wide on **tablet** (from 769px)
- `is-4-desktop`: Restricts the wrapper to be 4/12 columns wide on **desktop** (from 1024px)
- `is-3-widescreen`: Restricts the wrapper to be 3/12 columns wide on **widescreen** monitors (from 1216px)

Bulma is designed with mobile first in mind, so you don't need to add a modifier to your form wrapper. By default, it's 100% of the mobile device's width.

Append these modifiers to the `column`:

```
<div class="column is-5-tablet is-4-desktop is-3-widescreen">
  <form class="box">
    Login
  </form>
</div>
```

Resize your browser to see it in action! The column takes up the whole width, up to 768px. If a higher value is reached, it resizes at each breakpoint to maintain a reasonable width at all times.

You can now implement the form's content.

Implementing the form's content

The login form will be built with four fields:

- An **email** input
- A **password** input
- A "Remember me" **checkbox**
- A "Login" submit **button**

You will add a placeholder and a `required` attribute to some of the fields, which handle **form errors**, so you can display to the user why they failed to login.

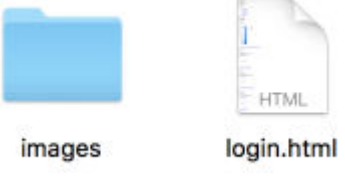
Logo

To reassure the user that they are indeed logging into the correct website, add a logo first. Replace the "Login" text you've had so far with your first field:

```
<form class="box">
  <div class="field has-text-centered">
```

```
    <img src="images/logo-bis.png" width="167">
  </div>
</form>
```

Note: Make sure the `images` folder is alongside your `login.html` file.



Bulma provides a `field` class that allows each form field to be *spaced* evenly. It also comes with helper classes like `has-text-centered` to center text and inline elements.

Email input

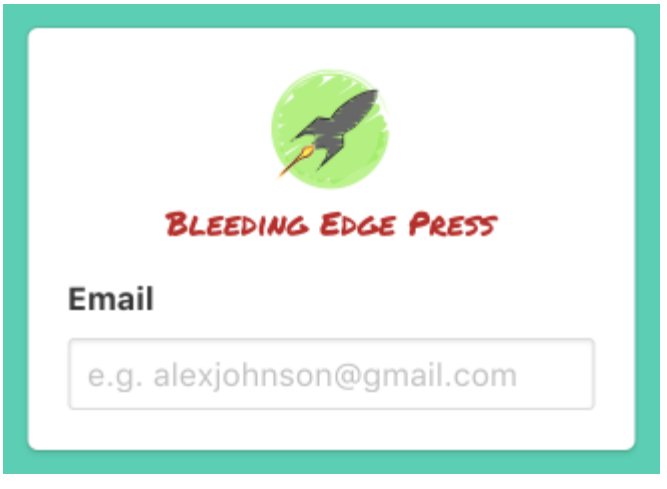
For the first input, use a couple of useful Bulma classes:

- `label`: Class for all form labels, which turns it **bold** and adds some space at the bottom.
- `control`: This class acts as a **wrapper** for the input, and will allow you to *enhance* it with icons.

After the first field, use the following HTML:

```
<div class="field">
  <label class="label">Email</label>
  <div class="control">
    <input class="input" type="email" placeholder="e.g. alexjohn-
```

```
son@gmail.com">
  </div>
</div>
```



Although you are using an HTML5 email input, decorate the input with an email icon from Font Awesome to hint at the content expected here.

In order to do that with Bulma, you must first add the `has-icons-left` modifier to the `control` wrapper. This is a Bulma modifier that adds some padding to the *left* of the wrapper to make room for an icon.

```
<div class="control has-icons-left">
```

You'll want to add the **envelope** Font Awesome icon and add modifiers so the icon is floated to the left and fits within the email input.

```
<span class="icon is-small is-left">
  <i class="fa fa-envelope"></i>
</span>
```

- `icon`: A Bulma element that defines an icon.
- `is-small`: A modifier that makes the icon small. You can also use the `is-large` modifier.
- `is-left`: Aligns the icon to the *left* of the form input.

The `control` wrapper now contains an input with an icon to the left.

```
<div class="control has-icons-left">
  <input class="input" type="email" placeholder="e.g. alex@smith.com" re-
```

```
quired>
  <span class="icon is-small is-left">
    <i class="fa fa-envelope"></i>
  </span>
</div>
```

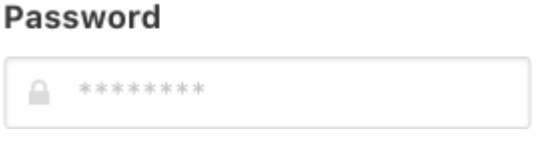
Note: Even if the icon loads *after* the page, the layout will not "jump" because Bulma makes sure that the space defined by the icon is fixed.

Password input

The password input is very similar to the email icon, so you can simply **duplicate** the first `field`, and modify a few parts:

- The **label** is now "Password"
- The input **type** is `password`
- The input **placeholder** is `********`
- The **icon** is `fa-lock`

```
<div class="field">
  <label class="label">Password</label>
  <div class="control has-icons-left">
    <input class="input" type="password" placeholder="********" required>
    <span class="icon is-small is-left">
      <i class="fa fa-lock"></i>
    </span>
  </div>
</div>
```



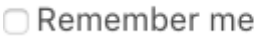
The same Bulma classes apply to this field as well.

Remember me checkbox

Add a simple `checkbox` for the "Remember me" feature. The `<label>` element allows you to **increase** the click zone of the checkbox: the text "Remember me" is clickable as well.

You don't need a `control` here since you aren't using an icon.

```
<div class="field">
  <label class="checkbox">
    <input type="checkbox">
    Remember me
  </label>
</div>
```



Login button

To complete your form, you only need a submit button. Bulma provides a `button` class that can be used on:

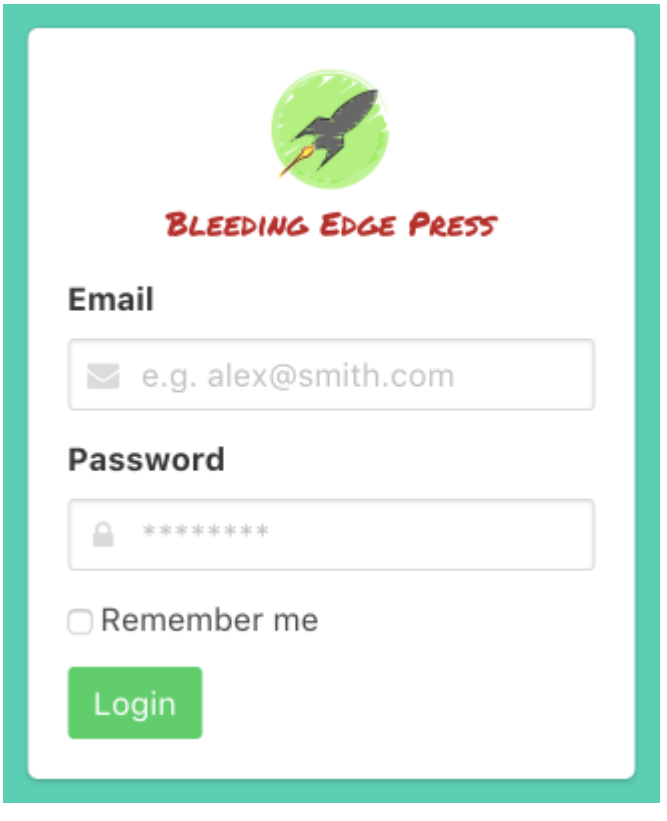
- anchor tags `<a>`
- button tags `<button>`
- input tags `<input type="submit">`

You'll want to use the `<button>` element since it's the most flexible and is a valid form element.

```
<div class="field">
  <button class="button is-success">
    Login
  </button>
</div>
```

Summary

The login page is now complete! Since you're using the `required` attributes for your email and password inputs, the form can only be submitted if these are valid.



Next up, you can move on to the portion the user reaches after logging in: the admin area.

Creating navigations and vertical menus

3

In the previous chapter, you learned how to create and control HTML forms with Bulma by creating a login form. Now it's time to build the admin area.

This chapter goes in-depth on how to use Bulma's `navigation` and `menu` components. These components (especially the navigation) are *essential* when creating a website. There's no need to reinvent the wheel each time, so let Bulma do all the heavy lifting. Remember, you can always modify Bulma's variables and adjust the user interface.

Note: To see the full code of the example used in this book take a look at the **book's accompanying GitHub page**.

In this example, it's safe to assume that the "user" is able to login correctly. Once "logged" in, an admin area should be displayed. The basic structure of the admin area is as follows:

- Dashboard
- Books
  - Book
- Customers
  - Customer
- Orders
  - Order

While each page will have its own specific content, some parts will be common across all templates. This includes the `navbar`, `menu` to the left, and the `main` content area to the right.

The first template you design is the Books template. Simply duplicate the `login.html` file, rename it to `books.html`, and remove everything inside the `<body>` so you only have the Doctype, the `<html>` tags, and the `<head>`.

Creating the navigation bar

Bulma comes with a flexible responsive `navbar`. You will use it to display a few elements:

- Company's logo, which will act as a home link
- Navigation's mobile burger icon
- Company's tagline
- User's name
- Dropdown menu with a few items: link to the user profile, a button to report a bug, and a link to sign out

```
<nav class="navbar has-shadow">
  <div class="navbar-brand">
    <!-- Logo, tagline, and navbar-burger -->
  </div>

  <div class="navbar-menu">
    <!-- User name, dropdown menu -->
  </div>
</nav>
```

The navigation's branding

In the navigation bar, you'll want to display the logo of the company. You want this visible at *all* times across all devices. There's no need to write any custom CSS. As you've probably guessed by now, you can use Bulma with component classes.

The `navbar-brand` lives on the *left* side of the navbar. It's always visible and can contain any number of `navbar-item`(s). It also holds the Bulma `navbar-burger`, which is used to toggle the `navbar-menu`.

For now, just add the logo `logo.png` and the tagline.

```
<div class="navbar-brand">
  <a class="navbar-item">
    <img src="images/logo.png">
  </a>
</div>
```

You don't need to specify the image dimensions, since Bulma makes sure that any `<img>` element residing in the `navbar-brand` will fit.

A navigation bar isn't very useful without any links or a way to access links. To do this, create three `<span>` tags. Each `<span>` tag will be a single line in the hamburger icon. If you try clicking on this now, it won't animate or do anything. To handle this, add the `navbar-burger` component class. This adds the styles needed to render a hamburger icon.

Let's add the `navbar-burger`, which is only displayed until the desktop breakpoint is reached (`1024px`):

```
<div class="navbar-brand">
  <a class="navbar-item">
    <img src="images/logo.png">
  </a>
  <div class="navbar-burger">
    <span></span>
    <span></span>
    <span></span>
  </div>
</div>
```

Now that the left part of the navbar brand is done, you can implement the right part.

The navigation's menu

Bulma's `navbar-menu` contains all of the other parts of your `navbar`. This part is visible when toggling the `navbar-burger`. The `navbar-menu` is hidden by default, however, it can be displayed by adding the `is-active` modifier.

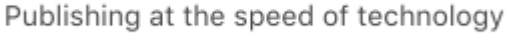
On the desktop and above, the `navbar-menu` is *always* visible, and fills up the remaining space left next to the `navbar-brand`.

The navbar menu itself is split into two parts:

- `navbar-start`: On the left (next to the `navbar-brand`)
- `navbar-end`: On the right

The left side is a good location for a tagline. Add this code after the `navbar-brand`, within the `navbar`:

```
<div class="navbar-menu">
  <div class="navbar-start">
    <div class="navbar-item">
      <small>Publishing at the speed of technology</small>
    </div>
  </div>
</div>
```



If you resize your browser, you'll notice that the tagline will only show up after you reach `1024px` in viewport width.

The Navigation's dropdown menu

Within the `navbar-menu`, as a sibling of `navbar-start`, you can now add the `navbar-end`, which will hold your dropdown menu.

```
<div class="navbar-end">

</div>
```

The `navbar-end`, should contain a dropdown menu when clicking on the navigation link. In this case, the navigation link is going to be the user, "Alex Johnson." You'll want a dropdown menu to appear when hovering over the user's name.

Since "Alex Johnson" is a link in the navigation, Bulma's `navbar-item` class is a perfect fit for this because it "defines" an item in the navbar. This item, as mentioned before, also displays a dropdown menu nested within it. You can append the `has-dropdown` modifier. This modifier *hides* the nested `navbar-dropdown` element unless hovered on.

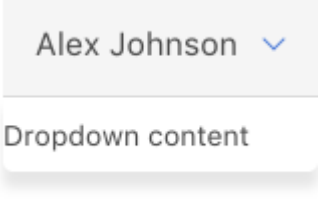
```
<div class="navbar-end">
  <div class="navbar-item has-dropdown">
    <div class="navbar-link">
      Alex Johnson
    </div>
    <div class="navbar-dropdown">
      Dropdown content
    </div>
  </div>
</div>
```

- `has-dropdown`: Modifier to the `navbar-item`. Hides the nested `navbar-dropdown` element.
- `navbar-link`: This will always be visible, and will act as the dropdown trigger.
- `navbar-dropdown`: Is the dropdown menu container.

The `navbar-dropdown` is hidden by default. You can either display it on hover, or with a CSS class toggle. For simplicity sake, it's easier to use the hover state. Simply add the `is-hoverable` modifier to the `navbar-item`:

```
<div class="navbar-item has-dropdown is-hoverable">
```

Hover over the "Alex Johnson" `navbar-item` to see the dropdown appear.

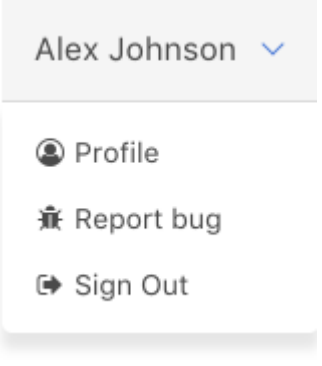


The `navbar-dropdown` can also contain `navbar-items`. You will need three items, each with a small icon.

Remove the text "Dropdown content" and add this content in it's place:

```
<div class="navbar-dropdown">
  <a class="navbar-item">
    <div>
      <span class="icon is-small">
        <i class="fa fa-user-circle-o"></i>
      </span>
      Profile
    </div>
  </a>
  <a class="navbar-item">
    <div>
      <span class="icon is-small">
        <i class="fa fa-bug"></i>
```

```
        </span>
        Report bug
      </div>
    </a>
    <a class="navbar-item">
      <div>
        <span class="icon is-small">
          <i class="fa fa-sign-out"></i>
        </span>
        Sign Out
      </div>
    </a>
  </div>
```



You now have a responsive `navbar` with all of the content required for your administration pages.

The main section

All of the admin pages are going to be split in two columns. The left column will contain the sidebar menu that will be common across all pages, while the right column's content will be specific to the current page.

Following the `navbar`, you can use Bulma's `section` element to wrap your main content:

```
<section class="section">
  <!-- The main content of the page -->
</section>
```

This provides the main content of the page some space, preventing it from reaching the edges of the viewport. You can now define your two-column layout.

Within this `section`, add the following:

```
<div class="columns">
  <div class="column is-4-tablet is-3-desktop is-2-widescreen">
    <!-- The sidebar -->
  </div>
  <div class="column">
    <!-- The right part, specific to each page -->
  </div>
</div>
```

Just like the login page, the first column will have a different size for each breakpoint. And because Bulma columns are automatically resized, the second column will fill up the remaining space. With the layout set up, you can now add the sidebar menu in the left column.

The sidebar menu

Much like the navigation in the previous section, Bulma's **menu** component acts in a very similar way. There are `menu` containers, `menu-lists`, and more.

Bulma provides a simple **menu** that can be used for any type of vertical navigation. In this case, you define links to navigate between the top-level content types: the dashboard, the books, the customers, and the orders.

This menu will live in the first `column` and will be to the *left* of the admin's user interface. To create a menu, create a `<nav>` element with the `menu` class.

```
<nav class="menu">

</nav>
```

You'll obviously want to add some more content and possibly give it a label. The `menu-label` class can be appended to any HTML element. This class, however, is most commonly used with things like paragraphs and headings.

Continuing the menu sidebar...

```
<nav class="menu">
  <p class="menu-label">
    Menu
```

```
    </p>
  </nav>
```

You'll also want a list for your menu. This will contain useful links to the Dashboard, Books, Customers, and Orders pages. The `menu-list` should be an unordered list with list items. This is no different than creating a standard navigation bar for a website.

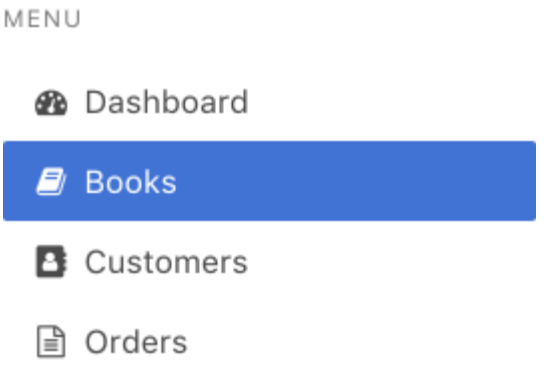
```
  <ul class="menu-list">
    <li>
      <a href="dashboard.html">
        <span class="icon">
          <i class="fa fa-tachometer"></i>
        </span>
        Dashboard
        </a>
    </li>
    <li>
      <a class="is-active" href="books.html">
        <span class="icon">
          <i class="fa fa-book"></i>
        </span>
        Books
      </a>
    </li>
    <li>
      <a href="customers.html">
        <span class="icon">
          <i class="fa fa-address-book"></i>
        </span>
        Customers
      </a>
    </li>
    <li>
      <a href="orders.html">
        <span class="icon">
          <i class="fa fa-file-text-o"></i>
        </span>
        Orders
      </a>
    </li>
  </ul>
```

Your final `menu` should resemble something close to this:

```
  <nav class="menu">
    <p class="menu-label">
      Menu
    </p>
    <ul class="menu-list">
```

```
      <li>
        <a href="dashboard.html">
          <span class="icon">
            <i class="fa fa-tachometer"></i>
          </span>
          Dashboard
          </a>
      </li>
      <li>
        <a class="is-active" href="books.html">
          <span class="icon">
            <i class="fa fa-book"></i>
          </span>
          Books
        </a>
      </li>
      <li>
        <a href="customers.html">
          <span class="icon">
            <i class="fa fa-address-book"></i>
          </span>
          Customers
        </a>
      </li>
      <li>
        <a href="orders.html">
          <span class="icon">
            <i class="fa fa-file-text-o"></i>
          </span>
          Orders
        </a>
      </li>
    </ul>
  </nav>
```

This will populate the left column with a vertical menu that will take up about one fourth of the page's width.



Since this is the `books.html` file, make sure to add the `is-active` modifier class on the appropriate menu item.

Summary

This template has all of the common parts for the site so far (the navbar and the section with the sidebar menu). Next, you will focus on the Books' specific content.

Creating responsive grids with common components

4

In this chapter, you will learn how to easily create responsive grids with Bulma. You will also learn how to add Bulma components to your user interface for common things like boxes, lists, and media groups, and learn how to create pagination with Bulma. This is all useful for creating large scale websites, like eCommerce websites.

Note: To see the full code of the example used in this book take a look at the **book's accompanying GitHub page**.

At this point, you already have your Bulma menu created in the left column. It's time to create a responsive grid that is the body of the right column. This same patten will be applied and repeated for the three content pages (books, customers, and orders). The user interface will follow the CRUD (Create Read Update Delete) pattern. For each type of content, you need the following UI components:

- A list to view all items
- An empty form to create an item
- A populated form to update an item previously created
- A button to delete an item

For the `books.html` template, the right column of the page will contain:

- Title
- Horizontal toolbar
- List of book items
- Pagination component

The toolbar

In the second column of the layout (the one with `column` only), let's begin by creating the meat of the body. You'll want to first add a `<h1>` and give it a class of `title`. Bulma's `title` class will make the text larger and bolder.

The toolbar is going to be *horizontal* and provides some extra options for users. To keep certain components inline with each other on the same *level* you should use the `level` component class.

Similarities between navbar and level

The `level` component acts very much like the `navbar` and its items. You should refrain from using the `navbar` classes in this case since your options are to primarily use a navigation bar.

Bulma's `level` follows a simple structure:

```
<nav class="level">
  <div class="level-left">
    <div class="level-item">
    </div>
  </div>
  <div class="level-left">
    <div class="level-item">
    </div>
  </div>
</div><!-- level -->
```

Creating the toolbar

At this point, you should be familiar with `navbar` and by extension `levels`. There are, however, a few modifier classes that this book hasn't gone over yet.

- `subtitle`: A subtitle. Has a different weight than `title`.
- `is-5`: A modifier for titles. Gives similar styles of a <h5>.
- `is-success`: A modifier that gives the element the "success" color. By default, "success" modifiers are green.
- `is-hidden-tablet-only`: Hides an element of tablet devices only.
- `select`: Much like `control` is used for inputs, `select` is used on <select> tags for consistent styling.

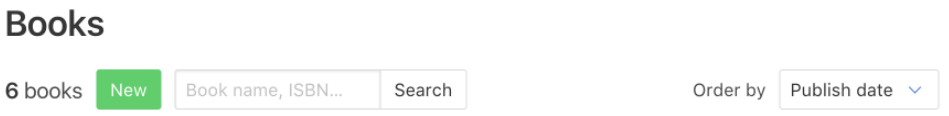
Your final HTML for the `level` bar should resemble something like this:

```
<h1 class="title">Books</h1>

<nav class="level">
  <div class="level-left">
    <div class="level-item">
      <p class="subtitle is-5">
        <strong>6</strong> books
```

```
      </p>
    </div>

    <p class="level-item">
      <a class="button is-success" href="new-book.html">New</a>
    </p>

    <div class="level-item is-hidden-tablet-only">
      <div class="field has-addons">
        <p class="control">
          <input class="input" type="text" placeholder="Book name, ISBN…">
        </p>
        <p class="control">
          <button class="button">
            Search
          </button>
        </p>
      </div>
    </div>
  </div>

  <div class="level-right">
    <div class="level-item">
      Order by
    </div>
    <div class="level-item">
      <div class="select">
        <select>
          <option>Publish date</option>
          <option>Price</option>
          <option>Page count</option>
        </select>
      </div>
    </div>
  </div>
</nav>
```

This adds a bold “Books” title, and a horizontal toolbar with several elements:

- Book count
- Green “New” button, that links to the page to create a new book
- Search box
- Sorting dropdown



Note: To prevent the toolbar from overflowing, the search box is hidden on tablets only. Thanks to the `level` class, all of the elements are vertically aligned and evenly spaced.

The books grid

To display all of the books sold by the publisher, you will define a two-dimensional grid of **six book items**. Each item will consist of:

- The book cover
- The name
- The price
- A list of meta data (number of pages, ISBN...)
- Links to edit and delete the book

To create the grid of the six books, you'll need to first create your standard `columns` row and give it six `<div>`s with the `column` class. Add an image *as a placeholder* for the book item:

```
<div class="columns">
  <div class="column">
    <img src="images/tensorflow.jpg" width="80">
  <div>
  <div class="column">
    <img src="images/tensorflow.jpg" width="80">
  <div>
  <div class="column">
    <img src="images/tensorflow.jpg" width="80">
  <div>
  <div class="column">
    <img src="images/tensorflow.jpg" width="80">
  <div>
  <div class="column">
    <img src="images/tensorflow.jpg" width="80">
  <div>
  <div class="column">
    <img src="images/tensorflow.jpg" width="80">
  <div>
</div>
```

Refresh this page in your browser of choice. At this point, you should see six book covers evenly spaced out in a single `columns` row. However, they are much too small and should probably be larger. Use modifiers to modify these columns to be different sizes on different devices.

To optimize the space, the number of columns will vary according to the viewport width:

- On mobile and tablet, there will be only 1 column
- On desktop, you will have 2 columns
- On widescreen, you will have 3 columns

```
<div class="column is-12-tablet is-6-desktop is-4-widescreen">
  <img src="images/tensorflow.jpg" width="80">
</div>
```

If you refresh your browser window now, you'll notice something *very* odd. Each of these book covers are the correct size depending on which device you're on, but...they're not "wrapping" to the next line as you might expect. That is because having a `columns` row will always automatically adjust the `column` width as seen before. Their modifier classes, however, are directly overriding and modifying the `column` width. Fortunately, there's a Bulma class that fixes this, so there's no need to create custom CSS.

That modifier class is `is-multiline`. Forgetting this class can be a common mistake that developers make when using Bulma for the first time. Please note that if you directly modify the width of a `column` and want them to wrap, you *need* the `is-multiline` class.

The book item

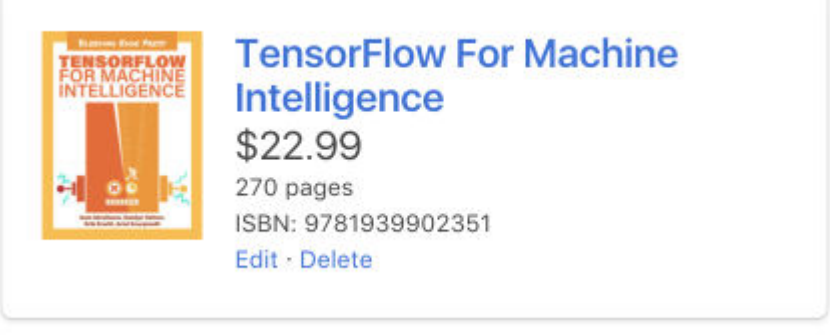
The Bulma box comes with a border and a shadow, which allows it to be visually distinct and separated. This is for a list of repeated items.

```
<article class="box">
  <div class="media">
    <aside class="media-left">
      <img src="images/tensorflow.jpg" width="80">
    </aside>

    <div class="media-content">
      <p class="title is-5 is-spaced is-marginless">
        <a href="edit-book.html">
          TensorFlow For Machine Intelligence
        </a>
      </p>
      <p class="subtitle is-marginless">
        $22.99
      </p>
```

```
        <div class="content is-small">
          270 pages
          <br>
          ISBN: 9781939902351
          <br>
          <a href="edit-book.html">Edit</a>
          <span>·</span>
          <a>Delete</a>
        </p>
      </div>
    </div>
  </article>
```

You'll notice that this HTML snippet contains a few more classes that this book hasn't gone over yet. One of these classes is `media`, which is repeatable, with nested content like book information or comments on a blog post.

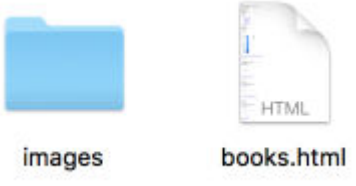
- `media`: Wrapper for nested, repeatable content.
- `media-left`: Much like, `navbar-left`, this is used for the left side of the `media` component.
- `media-content`: A wrapper for all the media's content.
- `is-marginless`: Removes any margin.
- `content`: Used for any *textual* content.



The `media` component is a very simple, but extremely useful UI pattern: it allows you to combine a small media element (like an image or an icon) with a larger bit of content side-by-side. By juxtaposing the book cover with its description, the book item is visually balanced, and optimizes the space.

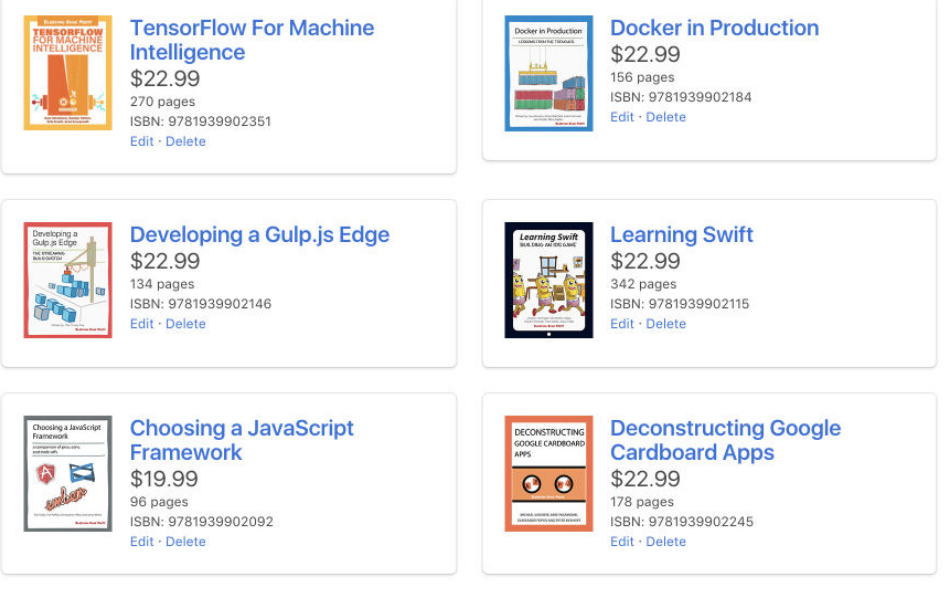
The title/subtitle combination emphasizes the book's most important information (name and price), while the `content` class is Bulma's default basic container for any longer piece of text.

Tip: For the image to appear, make sure to have the `images` folder alongside your `books.html` file.



Add the five other books and their respective images:

- “Docker in Production” -> `docker.jpg`
- “Developing a Gulp.js Edge” -> `gulp.jpg`
- “Learning Swift” -> `swift.jpg`
- “Choosing a JavaScript Framework” -> `js-framework.jpg`
- “Deconstructing Google Cardboard Apps” -> `google-cardboard.jpg`



Now that you have six items in your grid, resize your browser to see how the layout goes from one column to two and then three.

Pagination

Because the number of books is dynamic, it is highly probable that you will end up with more than six books (or twelve if you decide to show twelve books per page). To prepare for that case, you can use Bulma's responsive pagination component, which will allow your interface to handle any number of books.

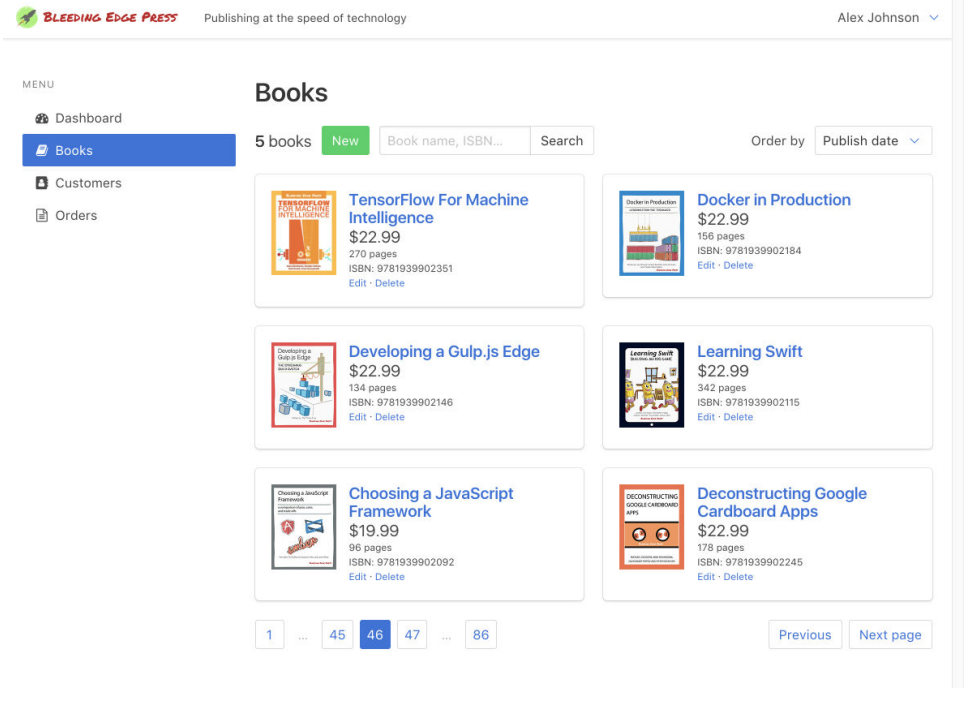
After the `columns is-multiline` element, add this snippet:

```
<nav class="pagination">
  <a class="pagination-previous">Previous</a>
  <a class="pagination-next">Next page</a>
  <ul class="pagination-list">
    <li>
      <a class="pagination-link">1</a>
    </li>
    <li>
      <span class="pagination-ellipsis">&hellip;</span>
    </li>
    <li>
      <a class="pagination-link">45</a>
    </li>
    <li>
      <a class="pagination-link is-current">46</a>
    </li>
    <li>
      <a class="pagination-link">47</a>
    </li>
    <li>
      <span class="pagination-ellipsis">&hellip;</span>
    </li>
    <li>
      <a class="pagination-link">86</a>
    </li>
  </ul>
</nav>
```

- `pagination`: Wrapper for pagination.
- `pagination-previous` and `pagination-next`: Used for incremental navigation.
- `pagination-link`: Each link in our pagination, which allows us to jump to different pages.
- `pagination-ellipsis`: Adds ellipsis for range separation.
- `is-current`: Highlights the current page.

Depending on the number of pages required by your UI, you can:

- Add or remove `pagination-ellipsis` elements
- Enable or disable the "Previous" and "Next" buttons by adding the `disabled` attribute



Summary

You have now completed the page that displays the *list* of books. Next, let's focus on the pages that will handle a *single* book.

Creating breadcrumbs and file upload fields

5

Continuing with what you've learned so far, let's create breadcrumbs and fields. This chapter is also going to build off of the previous chapters by creating the single book detail pages.

Note: To see the full code of the example used in this book take a look at the **book's accompanying GitHub page**.

There are two cases where a single book template will be used: to create a new book (`new-book.html`), and to edit an existing one (`edit-book.html`), since the delete action is simply a link in the list of books.

Duplicate the `books.html` file, rename it to `new-book.html`, and remove everything in the right column (`title`, `level`, `columns is-multiline`, and `pagination`), so only the `navbar` and the left sidebar `menu` remain.

New book detail template

The new book detail template is comprised of two components:

- A breadcrumb, to both tell the user where they are, and allow them to navigate back
- A form, to allow the user to input a book's information

Breadcrumb

The `new-book.html` page is reached by clicking on the green "New" button on `books.html`, and can thus be considered a *subpage* of the latter. To highlight this hierarchy to the user, you can display a Bulma breadcrumb:

```
<nav class="breadcrumb">
  <ul>
    <li>
      <a href="books.html">Books</a>
```

```
      </li>
      <li class="is-active">
        <a href="new-book.html">New book</a>
      </li>
    </ul>
  </nav>
```

The active item is black and not clickable, since it's the current page.

The book form

Each book will have the following fields:

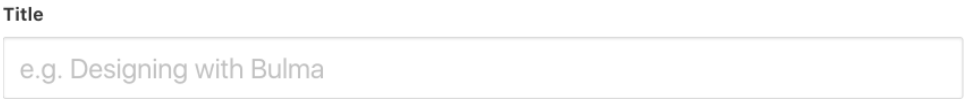
- title
- price
- page count
- ISBN
- cover image

Like the login page, the creation of a new book requires an HTML `<form>` that will use the following Bulma elements:

- label
- text input
- textarea
- file upload
- buttons

Right after the breadcrumb, create the form and add the first field:

```
<form>
  <div class="field">
    <div class="field">
      <label class="label">Title</label>
      <div class="control">
        <input class="input is-large" type="text" placeholder="e.g. Design-
ing with Bulma" required>
      </div>
    </div>
  </div>
</form>
```



Since the title is the most important information of a book, it uses a large input, thanks to the modifier class `is-large`. This input holds no value, since it's a creation form, and is required.

The following three fields are for the price, the page count, and the ISBN. Since these all hold relatively *short* values, they can be displayed as three columns on the desktop. Within the form, after the first field, add the following:

```
<div class="columns is-desktop">
  <div class="column">
    <label class="label">Price</label>
    <div class="control has-icons-left">
      <input class="input" type="number" placeholder="e.g. 22.99" required>
      <span class="icon is-small is-left">
        <i class="fa fa-dollar"></i>
      </span>
    </div>
  </div>

  <div class="column">
    <label class="label">Pages</label>
    <div class="control">
      <input class="input" type="number" placeholder="e.g. 270" required>
    </div>
  </div>

  <div class="column">
    <label class="label">ISBN</label>
    <div class="control">
      <input class="input" type="text" placeholder="e.g. 9781939902351" re-
quired>
    </div>
  </div>
</div>
```

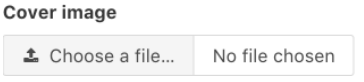
For this columns row, the modifier `is-desktop` is used to show the columns row *only* on desktop devices. This will hide the row on mobile and tablet sizes.

To specify the price currency, the `control` has a `has-icons-left` modifier, and contains an additional Bulma icon, which wraps the `fa-dollar` Font Awesome icon.

For the cover image, Bulma provides a file input, that holds an icon, a label, and an optional file name. After the columns, add another field:

```
<div class="field">
  <label class="label">Cover image</label>
  <div class="control">
    <div class="file has-name">
      <label class="file-label">
        <input class="file-input" type="file">
        <span class="file-cta">
          <span class="file-icon">
            <i class="fa fa-upload"></i>
          </span>
          <span class="file-label">
            Choose a file…
          </span>
        </span>
        <span class="file-name">
          No file chosen
        </span>
      </label>
    </div>
  </div>
</div>
```

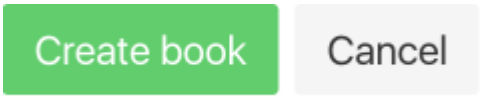
- `field`: Used on form fields to keep spacing consistent.
- `file`: An interactive wrapper of a file input. This is the container.
- `file-label:` The actual interactive and clickable part of the element.
- `file-input`: The native file input, hidden for styling purposes.
- `file-cta`: The upload call-to-action.
- `file-icon`: Optional upload icon.
- `file-name`: Container for the chosen file name.



The `file-name` element can be updated when the user chooses a file from their computer, but hasn't uploaded it yet.

Lastly, the form needs a couple of buttons: one to create the book (if all fields are populated), and one to cancel the creation. Bulma provides a `button` class that allows you to easily display a list of buttons:

```
<div class="field">
  <div class="buttons">
    <button class="button is-medium is-success">Create book</button>
    <button class="button is-medium is-light">Cancel</button>
  </div>
</div>
```



Edit the book template

The page to edit a book is almost *identical* to the one where you create a book. The only differences are:

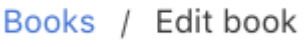
- The breadcrumb says "Edit book"
- All of the HTML `value` attributes are already populated
- The cover image is displayed
- The green button label says "Save changes" instead of "Create book"

As a result, you can simply duplicate the `new-book.html` file, rename it to `edit-book.html`, and apply a few changes.

In the breadcrumb, change "New book" to "Edit book":

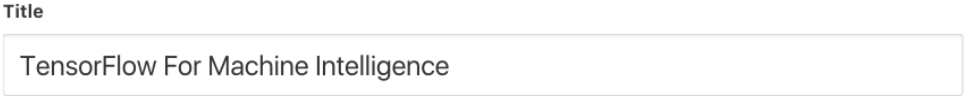
```
<nav class="breadcrumb">
  <ul>
    <li>
      <a href="books.html">Books</a>
    </li>
```

```
      <li class="is-active">
        <a href="edit-book.html">Edit book</a>
      </li>
    </ul>
  </nav>
```



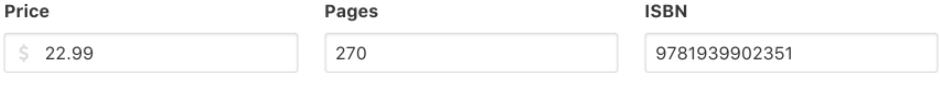
The first input's `value` attribute should hold a book title:

```
<input class="input is-large" type="text" placeholder="e.g. Designing with
Bulma" value="TensorFlow For Machine Intelligence" required>
```



The following three inputs should have a value as well. Find each `input`, and add a value:

```
<input class="input" type="number" placeholder="e.g. 22.99" value="22.99" re-
quired>
<input class="input" type="number" placeholder="e.g. 270" value="270" re-
quired>
<input class="input" type="text" placeholder="e.g. 9781939902351" val-
ue="9781939902351" required>
```



Since a cover image has already been uploaded at this point, it has to be displayed. You can simply use another `control` between the "Cover image" label, and the file input control:

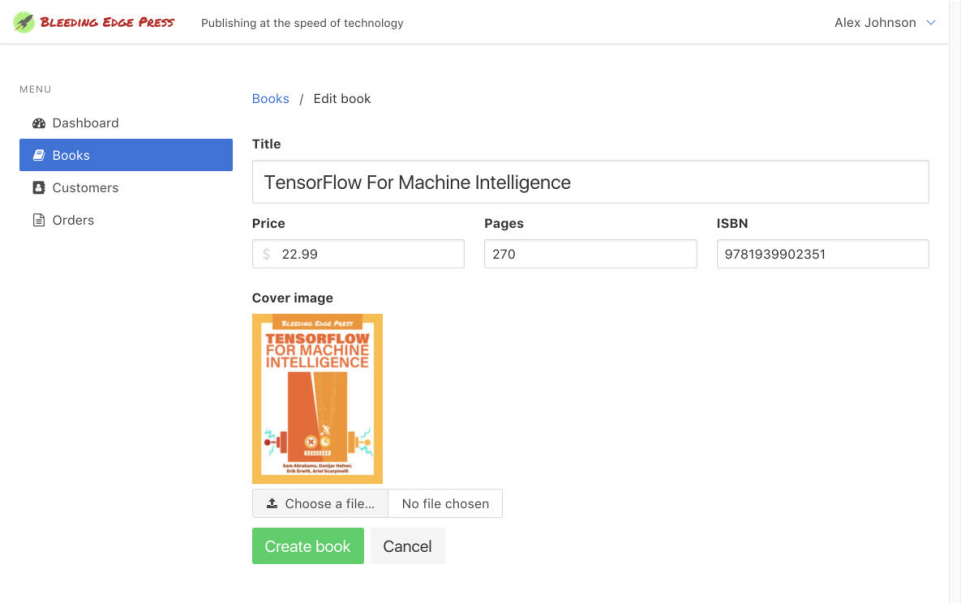
```
<div class="field">
  <label class="label">Cover image</label>
  <div class="control">
    <img src="images/tensorflow.jpg">
```

```
  </div>
  <div class="control">
    <div class="file has-name">
<!-- etc. -->
```

Cover image

Choose a file... No file chosen

This UI prevents the user from deleting the cover image without uploading a new one.



Summary

The Book templates are done! You can now focus on the Customers content type.

Creating tables and selecting dropdowns

6

Like previous chapters, this one will be continuing the project that you've been building. This chapter will highlight tables, illustrating how you can easily create tables with the classes provided.

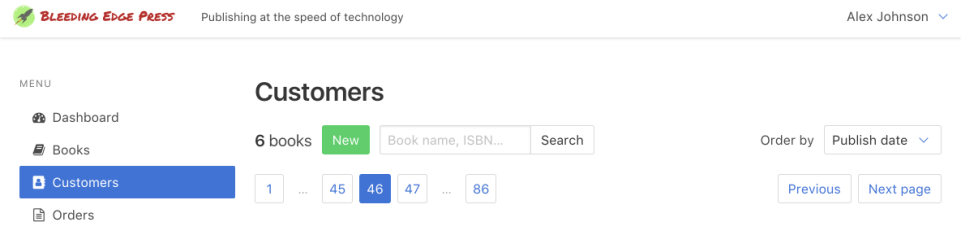
Note: To see the full code of the example used in this book take a look at the **book's accompanying GitHub page**.

After having implemented the three **Book** templates required for basic CRUD functionality, you can now focus on the next content type: **Customers**. The functionalities will actually be identical: creating customers, editing/viewing them, and eventually deleting them. The differences will be in the fields required for a customer, and the way the list of customers will be displayed: instead of using a grid of boxes, the customers will be displayed in a Bulma `<table>`.

The list of customers

Duplicate `books.html`, rename it to `customers.html`, and perform a few small changes:

- Move the `is-active` class in the sidebar menu from "Books" to "Customers"
- Rename the title from "Books" to "Customers"
- Remove the grid of book items

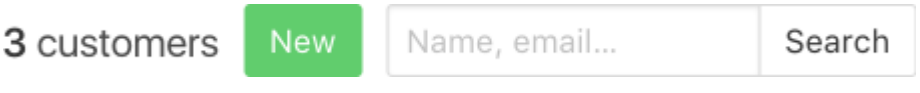


As you can see, this page still needs to be updated quite a bit.

Updating the toolbar

The toolbar residing in the `level` component only requires some text replacements:

- "6 books" is now "3 customers"
- The "New" button target is now `new-customer.html`
- The "Book name, ISBN..." placeholder is now "Name, email..."



The `level-right` will now contain toggle elements instead of a dropdown. Replace it with the following:

```
<div class="level-right">
  <p class="level-item"><strong>All</strong></p>
  <p class="level-item"><a>With orders</a></p>
  <p class="level-item"><a>Without orders</a></p>
</div>
```

By simply having one `<strong>` and two `<a>` elements, you have a UI for very basic toggle controls.

Implementing the table of customers

To keep the UI simple, each customer will have:

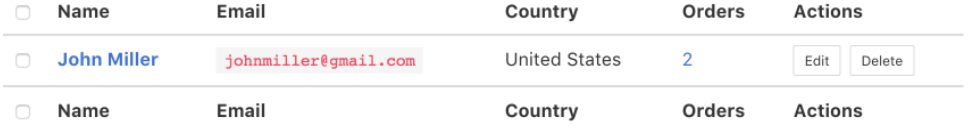
- A name
- An email address
- An address with street name, postcode, city, and country
- A list of orders

Since there is no image to display, let's use a Bulma `<table>` here to have a higher density of information.

Between the `level` and the `pagination`, add the following:

```
<table class="table is-hoverable is-fullwidth">
  <thead>
    <tr>
      <th class="is-narrow">
        <input type="checkbox">
      </th>
      <th>Name</th>
      <th>Email</th>
      <th>Country</th>
      <th>Orders</th>
      <th>Actions</th>
    </tr>
  </thead>
  <tfoot>
    <tr>
      <th class="is-narrow">
        <input type="checkbox">
      </th>
      <th>Name</th>
      <th>Email</th>
      <th>Country</th>
      <th>Orders</th>
      <th>Actions</th>
    </tr>
  </tfoot>
  <tbody>
    <tr>
      <td>
        <input type="checkbox">
      </td>
      <td>
        <a href="edit-customer.html">
          <strong>John Miller</strong>
        </a>
      </td>
```

```
        <td><code>johnmiller@gmail.com</code></td>
        <td>United States</td>
        <td>
          <a href="customer-orders.html">2</a>
        </td>
        <td>
          <div class="buttons">
            <a class="button is-small" href="edit-customer.html">Edit</a>
            <a class="button is-small">Delete</a>
          </div>
        </td>
      </tr>
    </tbody>
</table>
```

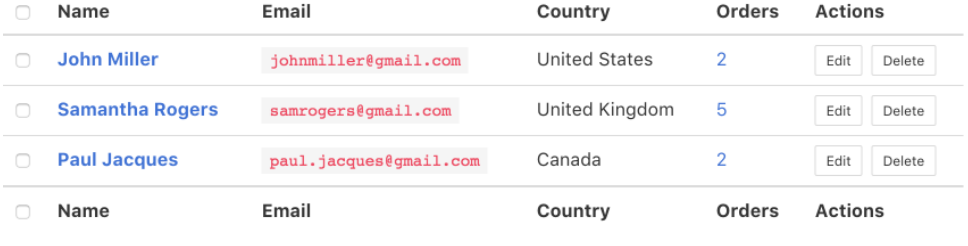


Note: Since the address can be very long, the country is sufficient for the list view.
The table uses two modifiers classes:

- `is-hoverable`: highlights the whole row when hovered
- `is-fullwidth`: forces the table to use the whole width available

The cell containing the checkbox is using the `is-narrow` Bulma modifier to make sure it only uses the minimum width required. This checkbox is often seen in tables for bulk edit functionalities.

Add two other rows, with other names, email addresses, countries, and a number of orders.



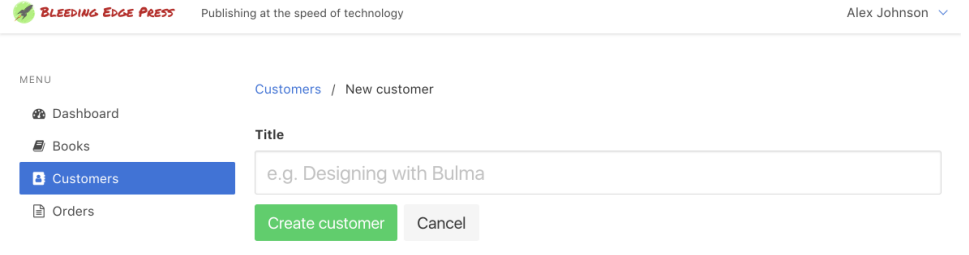
The customers template is ready. You can now focus on the single customer templates.

New customer template

The new customer template has the same structure as the new book one: a breadcrumb and a list of form fields.

Duplicate the `new-book.html` file, and rename it to `new-customer.html`. In the sidebar menu, move the `is-active` class to the "Customers" item. In the right column's breadcrumb, change any instance of "book" to "customer."

You can now focus on the `<form>`. Remove all fields, except the first large input, and the last set of buttons.



The first field can simply be repurposed by changing the label and placeholder.

```
<div class="field">
  <div class="field">
    <label class="label">Full name</label>
    <div class="control">
      <input class="input is-large" type="text" placeholder="e.g. Alex
Smith" required>
    </div>
  </div>
</div>
```

The second field is an email one, with an envelope icon.

```
<div class="field">
  <label class="label">Email</label>
  <div class="control has-icons-left">
    <input class="input" type="email" placeholder="e.g. alexjohn-
son@gmail.com" required>
```

```
      <span class="icon is-small is-left">
        <i class="fa fa-envelope"></i>
      </span>
    </div>
  </div>
```

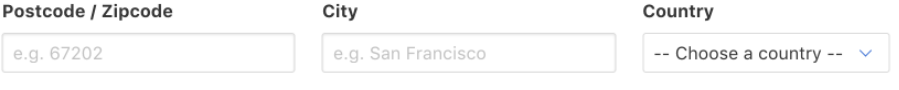
The third and fourth fields are for the customer's address. Only the first line is required. Note how the second line doesn't require a label.

```
<div class="field">
  <label class="label">Address</label>
  <div class="control">
    <input class="input" type="text" placeholder="Number and street name" re-
quired>
  </div>
</div>

<div class="field">
  <div class="control">
    <input class="input" type="text" placeholder="Second address line (op-
tional)">
  </div>
</div>
```

For the postcode/city/country combination, save space by using a set of Bulma columns.

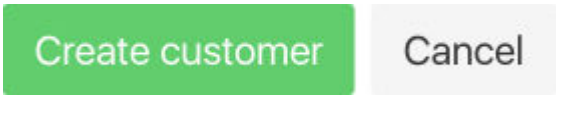
```
<div class="columns is-multiline">
  <div class="column is-12-tablet is-6-tablet is-4-desktop">
    <label class="label">Postcode / Zipcode</label>
    <div class="control">
      <input class="input" type="text" placeholder="e.g. 67202" required>
    </div>
  </div>
```

```
  <div class="column is-12-tablet is-6-tablet is-4-desktop">
    <label class="label">City</label>
    <div class="control">
      <input class="input" type="text" placeholder="e.g. San Francisco" re-
quired>
    </div>
  </div>

  <div class="column is-12-tablet is-6-tablet is-4-desktop">
    <label class="label">Country</label>
    <div class="control">
      <div class="select">
        <select>
          <option>-- Choose a country --</option>
          <option>Canada</option>
          <option>United Kingdom</option>
          <option>United States</option>
        </select>
      </div>
    </div>
  </div>
</div>
```

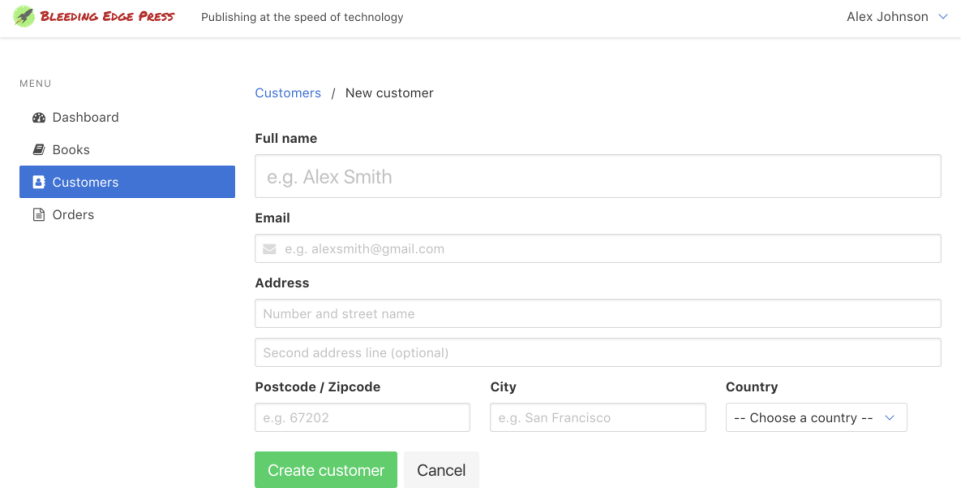


The last set of buttons only needs to be renamed.

```
<div class="field">
  <div class="buttons">
    <button class="button is-medium is-success">Create customer</button>
    <button class="button is-medium is-light">Cancel</button>
  </div>
</div>
```



The whole page is ready:



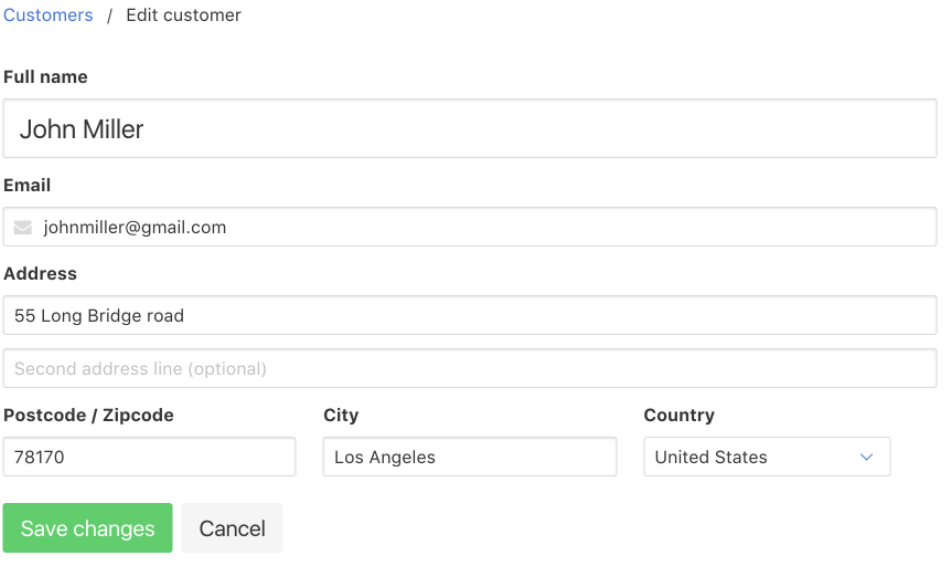
You now have a “Create customer” form, that can be re-used for the “Edit customer” template.

Edit customer template

As with the `edit-book.html` template, the “Edit customer” is simply the “New customer” template, but populated with values.

Duplicate the `new-customer.html` file, rename it to `edit-customer.html`, and apply a few changes:

- Put “Edit customer” in the breadcrumb
- Add a value for each `required` field
- Choose a country
- Rename the green button to “Save changes”



For the country selection, simply add the `selected` HTML attribute:

```
<select>
  <option>-- Choose a country --</option>
  <option>Canada</option>
  <option>United Kingdom</option>
  <option selected>United States</option>
</select>
```

Summary

You have now learned how to make basic tables, and in the next chapter you will learn how to make more advanced tables.

Creating more tables and selecting dropdowns

7

This chapter is going to continue using tables in dropdowns, with a focus on more advanced cases. At this point, if you've followed along, you have created the majority of the application. There are, however, a few more things to do.

Note: To see the full code of the example used in this book take a look at the **book's accompanying GitHub page**.

The *Order* content type connects a *Customer* to one or multiple *Books*. Each *Order* will have:

- An id number
- An associated customer
- A date
- A list of books
- A status, one of "In progress", "Successful", or "Failed"
- A total cost

List of orders

To display the list of orders, you can use a similar table to the customers table.

Duplicate `customers.html`, rename it `orders.html`, and perform a few changes:

- Move the `is-active` class in the sidebar menu
- Change the title to "Orders"
- Write "2 orders" instead of "3 customers"
- Remove the "New" button
- Change the search placeholder to "Order #, customer..."

The green "New" button is removed because the UI assumes that an order is **automatically** created when a customer purchases a book on the publisher's website.

The table only requires new columns:

- order #
- customer
- date
- number of books
- status
- total cost

```
<table class="table is-hoverable is-fullwidth">
  <thead>
    <tr>
      <th>Order #</th>
      <th>Customer</th>
      <th>Date</th>
      <th>Books</th>
      <th>Status</th>
      <th class="has-text-right">Total</th>
    </tr>
  </thead>
  <tfoot>
    <tr>
      <th>Order #</th>
      <th>Customer</th>
      <th>Date</th>
      <th>Books</th>
      <th>Status</th>
      <th class="has-text-right">Total</th>
    </tr>
  </tfoot>
  <tbody>
    <tr>
      <td>
        <a href="edit-order.html"><strong>787352</strong></a>
      </td>
      <td>
        <a href="edit-customer.html">John Miller</a>
      </td>
```

```
      <td>Nov 18, 17:38</td>
      <td>2</td>
      <td>
        <span class="tag is-warning">In progress</span>
      </td>
      <td class="has-text-right">$56.98</td>
    </tr>
  </tbody>
</table>
```

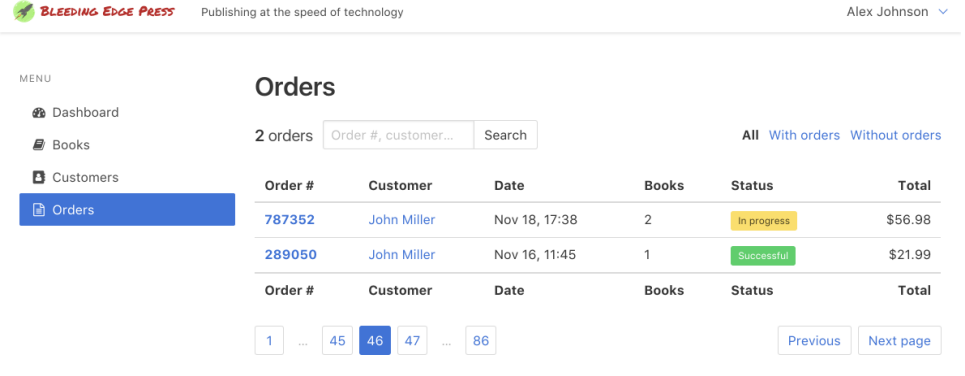
- `has-text-right`: Aligns the text in the element to the right.
- `tag`: A Bulma component. Renders a small colored element to help relay information.
- `is-warning`: A modifier used to assign the "warning" color. In this case, the default yellow color.

Order #	Customer	Date	Books	Status	Total
787352	John Miller	Nov 18, 17:38	2	In progress	$56.98
Order #	**Customer**	**Date**	**Books**	**Status**	**Total**

The order status uses a Bulma **tag**. Each status can have its own modifier:

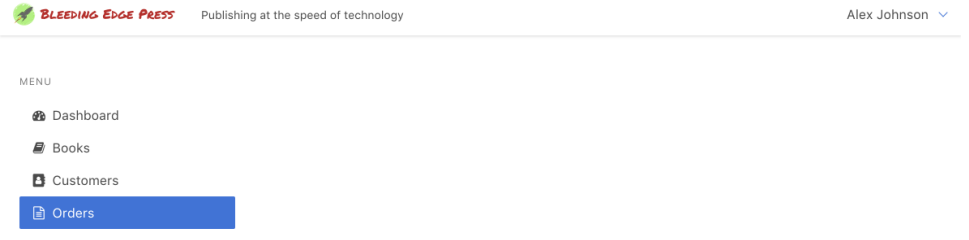
- In progress -> `is-warning`
- Successful -> `is-success`
- Failed -> `is-danger`

Add another order, with different data. Your template is now complete.



Edit Order

Duplicate `orders.html`, rename it to `edit-order.html`, and remove everything in the right hand column.



Each order comes with a unique auto-generated id that can be displayed as the main title, just below the breadcrumb.

```
<nav class="breadcrumb">
  <ul>
    <li>
      <a href="orders.html">Orders</a>
    </li>
    <li class="is-active">
      <a>Edit order</a>
    </li>
  </ul>
</nav>

<h1 class="subtitle is-3">
  <span class="has-text-grey-light">Order</span> <strong>787352</strong>
</h1>
```

Since an order connects a customer to a list of books, the logical way to display this relationship is with two columns. Following the subtitle, add this snippet:

```
<div class="columns is-desktop">
  <div class="column is-4-desktop is-3-widescreen">
    <!-- Left column, for order information and customer -->
  </div>
  <div class="column">
    <!-- Right column, for the list of books -->
  </div>
</div>
```

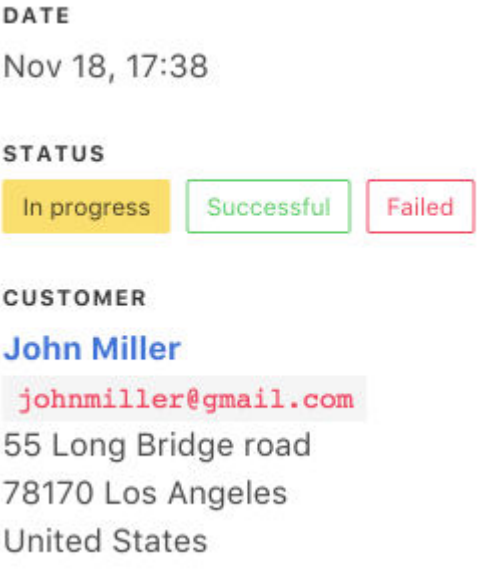
Order information

Most of the information displayed here is read-only. The only changeable element is the order status.

In the left column, add this code:

```
<p class="heading">
  <strong>Date</strong>
</p>
<p class="content">
  Nov 18, 17:38
</p>

<p class="heading">
  <strong>Status</strong>
</p>
<div class="buttons">
  <button class="button is-small is-warning">In progress</button>
  <button class="button is-small is-success is-outlined">Successful</button>
  <button class="button is-small is-danger is-outlined">Failed</button>
</div>

<p class="heading">
  <strong>Customer</strong>
</p>
<p class="content">
  <strong>
    <a href="edit-customer.html">John Miller</a>
  </strong>
  <br>
  <code>johnmiller@gmail.com</code>
  <br>
  55 Long Bridge road
  <br>
  78170 Los Angeles
  <br>
```

```
    United States
  </p>
```



The three buttons act as a mutually exclusive list, where the `is-outlined` items are inactive, while the third one is selected.

The customer's name has a link to its editing page, in case the user has to update the customer's address, for example, while viewing the order.

List of books

While the previous list of books (in `books.html`) was a grid of boxes, this list of books chosen by the customer doesn't need to be as detailed. As a matter of fact, since this list is editable, a Bulma **table** is the best UI choice here.

In the right column, add this snippet:

```
<p class="heading">
  <strong>Books</strong>
</p>
<table class="table is-bordered is-fullwidth">
  <thead>
    <tr>
      <th class="is-narrow">Cover</th>
      <th>Title</th>
```

```
        <th class="has-text-right is-narrow">Price</th>
        <th class="has-text-right is-narrow">Amount</th>
        <th class="has-text-right is-narrow">Total</th>
      </tr>
    </thead>
    <tfoot>
      <tr>
        <th colspan="5" class="has-text-right">$42.98</th>
      </tr>
    </tfoot>
    <tbody>
      <tr>
        <td>
          <img src="images/tensorflow.jpg" width="40">
        </td>
        <td>
          <a href="edit-book.html">
            <strong>
              TensorFlow For Machine Intelligence
            </strong>
          </a>
        </td>
        <td class="has-text-right">
          $22.99
        </td>
        <td class="has-text-right">
          <input class="input is-small" type="number" value="1" maxlength="2"
max="2">
        </td>
        <td class="has-text-right">
          $22.99
        </td>
      </tr>
      <tr>
        <td>
          <img src="images/js-framework.jpg" width="40">
        </td>
        <td>
          <a href="edit-book.html">
            <strong>
              Choosing a JavaScript Framework
            </strong>
          </a>
        </td>
        <td class="has-text-right">
          $19.99
        </td>
        <td class="has-text-right">
          <input class="input is-small" type="number" value="1" maxlength="2"
max="2">
```

```
        </td>
        <td class="has-text-right">
          $19.99
        </td>
      </tr>
    </tbody>
  </table>
```

BOOKS

Cover	Title	Price	Amount	Total
	TensorFlow For Machine Intelligence	$22.99	1	$22.99
	Choosing a JavaScript Framework	$19.99	1	$19.99
				$42.98

Each row is a book purchased by the customer. It links to the book itself, and the amount is editable.

The last row sums up the cost.

Row Form

There are many reasons why an order would need to be altered:

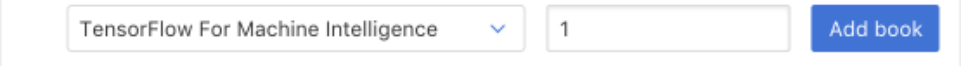
- A book ran out of stock
- The customer wants the same book twice instead of once
- The customer purchased the wrong book and wants to replace it
- Another book is added to the order

This is why the amount of books is editable, but the user needs a way to *add* a book that is not in the list yet.

Add a small form as the last row in the `tbody`:

```
<tr>
  <td colspan="5">
```

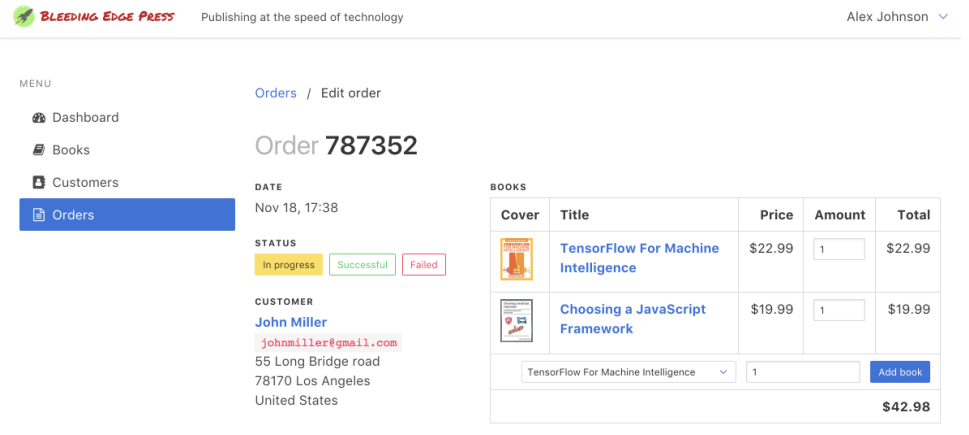
```
    <div class="field is-grouped is-grouped-right">
      <div class="control">
        <div class="select is-small">
          <select>
            <option>TensorFlow For Machine Intelligence</option>
            <option>Docker in Production</option>
            <option>Developing a Gulp.js Edge</option>
            <option>Learning Swift</option>
            <option>Choosing a JavaScript Framework</option>
            <option>Deconstructing Google Cardboard Apps</option>
          </select>
        </div>
      </div>
      <div class="control">
        <input class="input is-small" type="number" value="1" placehold-
er="Amount" maxlength="2" max="2">
      </div>
      <div class="control">
        <a class="button is-small is-link">Add book</a>
      </div>
    </div>
  </td>
</tr>
```



For such a small horizontal form, a Bulma **grouped field** is used: it combines multiple `control` elements on a single line, thanks to the `field is-grouped` class combination.

Summary

The template is now complete.



It is time to revisit the top-level template: the Dashboard.

8 Creating notifications and cards

At this point, you've explored a decent amount of what the Bulma framework has to offer. There are a lot of component and modifier classes that you can choose from. We hope you can see how you are able to create clean and structured user interfaces *without custom CSS code.* That's pretty cool. Of course, you can always modify Bulma with your own variables or add your own custom styles.

There are a few aspects of Bulma that this book hasn't explored yet: `notification`s and `card`s. Let's wrap up the application, and in later chapters of the book, you'll learn about using Bulma with Vanilla JavaScript as well as the Angular, Vue, and React frameworks.

Note: To see the full code of the example used in this book take a look at the **book's accompanying GitHub page**.

The dashboard is the page the user lands on after logging in. It is usually the last page to be designed because it acts as both a summary of and a shortcut to the other pages of the admin area. Hence, why building the dashboard is the last step of this chapter; the idea is to take content of the other pages, and present them in a succinct way.

The layout will be a grid of components, each of them related to one or multiple content types:

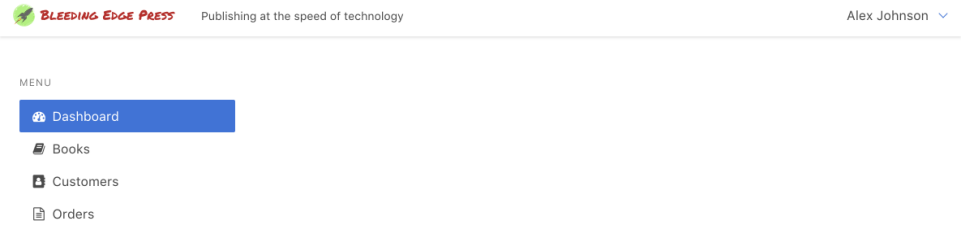
- The most important metrics
- A list of the latest orders
- The most popular books
- The most loyal customers

By using standard Bulma components, you can easily build a dashboard with a wide range of use cases.

Title, time range

The dashboard's main purpose is to provide a rapid **overview** within a certain **timeframe**, so that the user can, at a glance, get a grasp of the state of the admin area.

Duplicate `books.html` and remove everything in the main right column (from the "Books" title to the pagination), so only the navbar at the top and the sidebar menu on the left remain. Move the `is-active` class as well:



In this now empty right column, start with a `level` component that will combine a title and a select dropdown.

```
<div class="level">
  <div class="level-left">
    <h1 class="subtitle is-3">
      <span class="has-text-grey-light">Hello</span> <strong>Alex Johnson</
strong>
    </h1>
  </div>
  <div class="level-right">
    <div class="select">
      <select>
        <option>Today</option>
        <option>Yesterday</option>
        <option>This Week</option>
        <option selected>This Month</option>
        <option>This Year</option>
        <option>All time</option>
      </select>
    </div>
  </div>
</div>
```

- `has-text-grey-light`: A helper class for typography. Assigns the text with a light grey color.

The title mentions the user's name, which acts as a confirmation after the user has logged in.

The right part has a select dropdown that allows the user to change the timeframe of the dashboard they are viewing (similarly to most analytics dashboard).

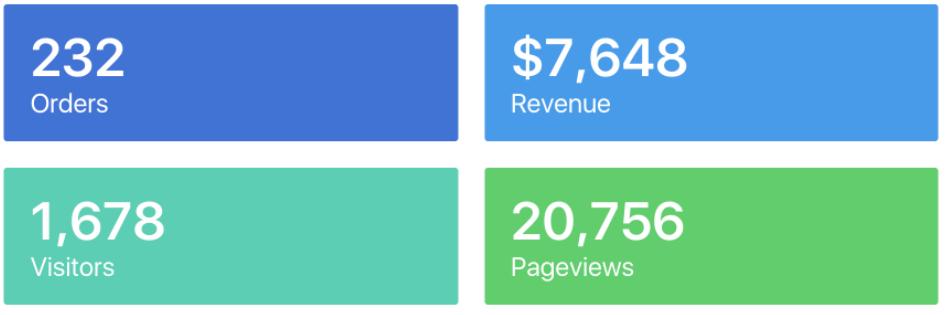
Important metrics

The dashboard is a transient page: the user sees it, has a rapid look-around, and navigates to the part that caught their attention. That is why the UI should provide information almost **instantly**.

Bulma provides `notification` elements that come in various colors. Combined with titles with a bigger font size, they are the perfect candidates for high-level metrics.

After the `level` component, add these columns:

```
<div class="columns is-multiline">
  <div class="column is-12-tablet is-6-desktop is-3-widescreen">
    <div class="notification is-link has-text">
      <p class="title is-1">232</p>
      <p class="subtitle is-4">Orders</p>
    </div>
  </div>

  <div class="column is-12-tablet is-6-desktop is-3-widescreen">
    <div class="notification is-info has-text">
      <p class="title is-1">$7,648</p>
      <p class="subtitle is-4">Revenue</p>
    </div>
  </div>

  <div class="column is-12-tablet is-6-desktop is-3-widescreen">
    <div class="notification is-primary has-text">
      <p class="title is-1">1,678</p>
      <p class="subtitle is-4">Visitors</p>
    </div>
  </div>

  <div class="column is-12-tablet is-6-desktop is-3-widescreen">
    <div class="notification is-success has-text">
      <p class="title is-1">20,756</p>
      <p class="subtitle is-4">Pageviews</p>
    </div>
  </div>
</div>
```



The columns are multiline so you can have one column on mobile and tablet, two on desktop, and four on widescreen.

Latest orders

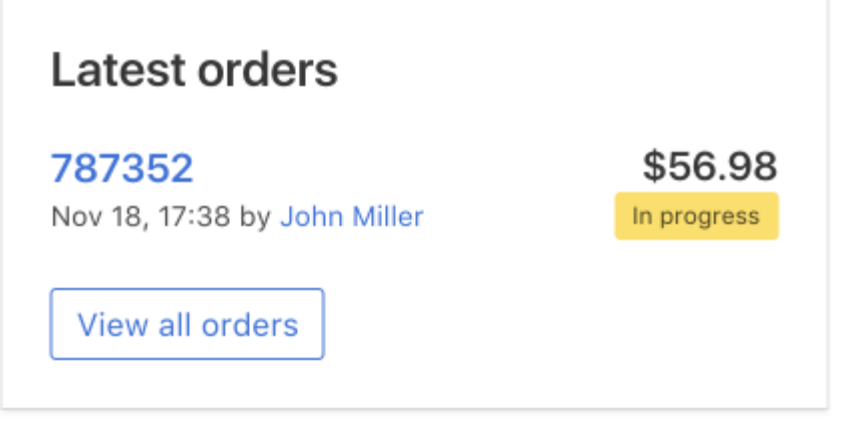
The orders is the content type that is most likely to be frequently populated, since they come from the website. That is why it makes sense to show its latest state right away, before navigating to the "Orders" page.

Because the columns implemented for the high-level metrics are multiline, you can simply append more column items at the end.

Right after the last `<div class="column is-12-tablet is-6-desktop is-3-widescreen">`, but still within the `<div class="columns is-multiline">`, add this new column:

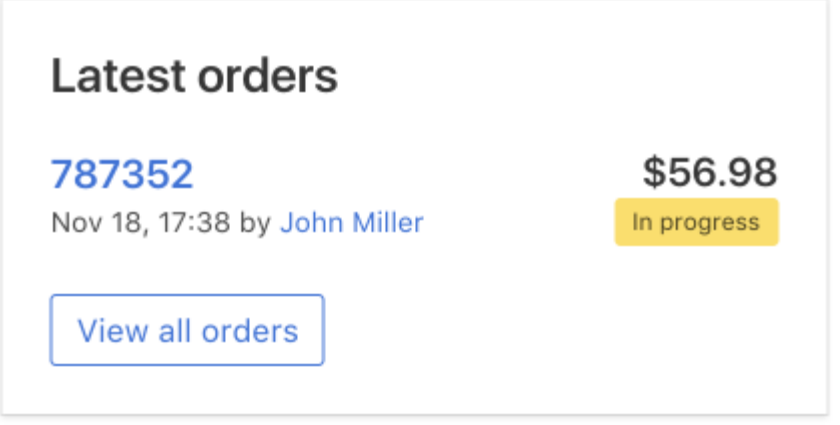
```
<div class="column is-12-tablet is-6-desktop is-4-fullhd">
  <div class="card">
    <div class="card-content">
      <h2 class="title is-4">
        Latest orders
      </h2>

      <div class="level">
        <div class="level-left">
          <div>
            <p class="title is-5 is-marginless">
              <a href="edit-order.html">787352</a>
            </p>
            <small>
              Nov 18, 17:38 by <a href="edit-customer.html">John Miller</a>
            </small>
          </div>
        </div>
```

```
        <div class="level-right">
          <div class="has-text-right">
            <p class="title is-5 is-marginless">
              $56.98
            </p>
            <span class="tag is-warning">In progress</span>
          </div>
        </div>
      </div>

      <a class="button is-link is-outlined" href="orders.html">View all or-
ders</a>
    </div>
  </div>
</div>
```



The `level` component here allows you to save vertical space by displaying the order id, date, and customer on the left, and push the total and status to the right.

Between the first `level` and the “View all orders” button, add a couple of other orders in this list, with different data:



Most popular books with cards

In this section you'll be creating cards. Cards are a Bulma component that are *great* at conveying information in a smaller amount of space. Usually, cards have visual elements with them, like an image or a video. Cards are *very* common and are especially common with eCommerce websites. However, let's create some cards with our in-progress book application.

Basic structure of a card

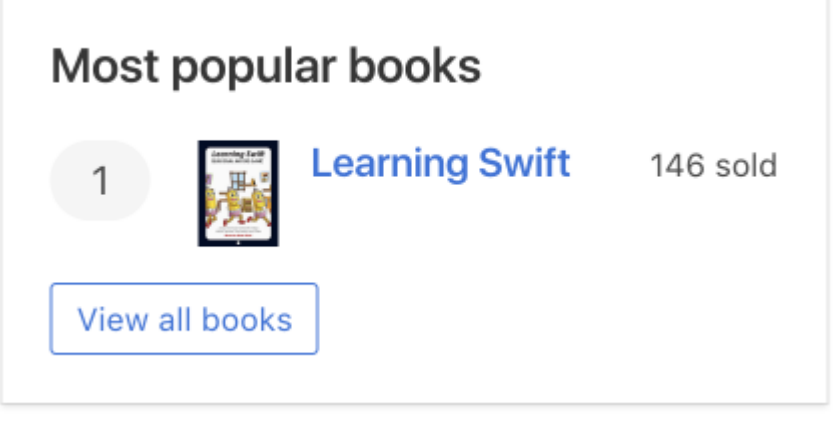
```
<div class="card">
  <div class="card-image">
    <!-- image here -->
  <div>
  <div class="card-content">
    <!-- content here -->
  <div>
</div>
```

For this example, you'll be using Bulma's `media` component for the card's content. Let's move on.

The dashboard should contain items that are likely to change over time. The list of the most popular books is one such item.

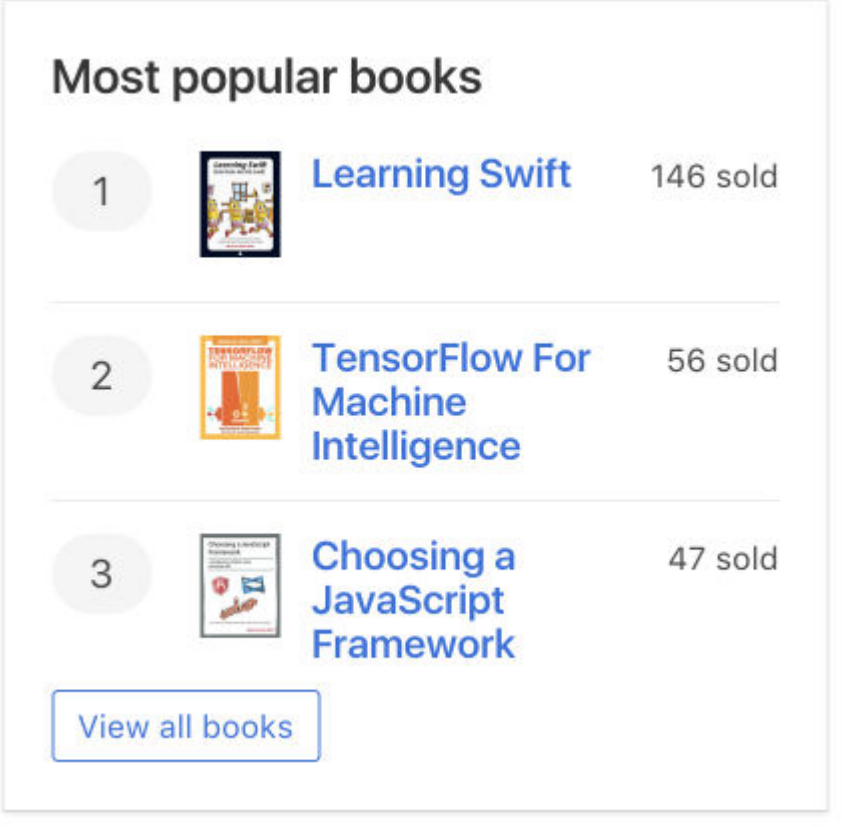
You can reuse the same layout as the previous column, but use a `media` component instead of a `level` one:

```
<div class="column is-12-tablet is-6-desktop is-4-fullhd">
  <div class="card">
    <div class="card-content">
      <h2 class="title is-4">
        Most popular books
      </h2>

      <div class="media">
        <div class="media-left is-marginless">
          <p class="number">1</p>
        </div>
        <div class="media-left">
          <img src="images/swift.jpg" width="40">
        </div>
        <div class="media-content">
          <p class="title is-5 is-spaced is-marginless">
            <a href="edit-book.html">Learning Swift</a>
          </p>
        </div>
        <div class="media-right">
          146 sold
        </div>
      </div>

      <a class="button is-link is-outlined" href="books.html">View all
books</a>
    </div>
  </div>
</div>
```



Two `media-left` elements are used here, which allows the UI to place multiple narrow elements side-by-side (the ranking and the cover image).

Now add a second and third book to the list:



Most loyal customers

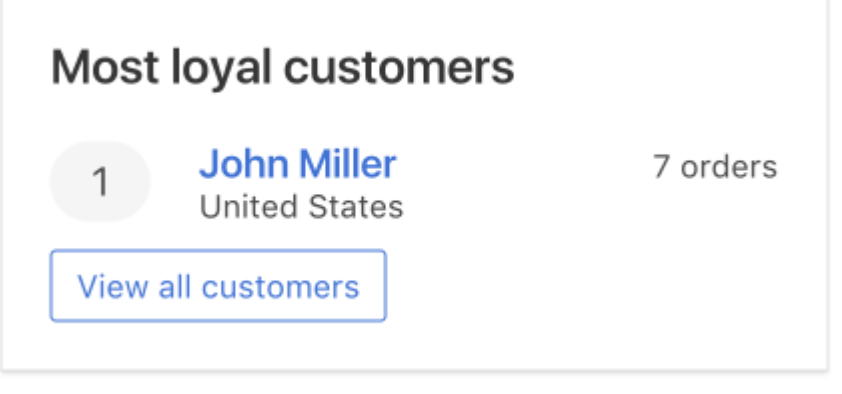
For the final column, you can provide an overview of the last content type: customers.

Right after the previous column, add the following:

```
<div class="column is-12-tablet is-6-desktop is-4-fullhd">
  <div class="card">
    <div class="card-content">
      <h2 class="title is-4">
        Most loyal customers
      </h2>
```

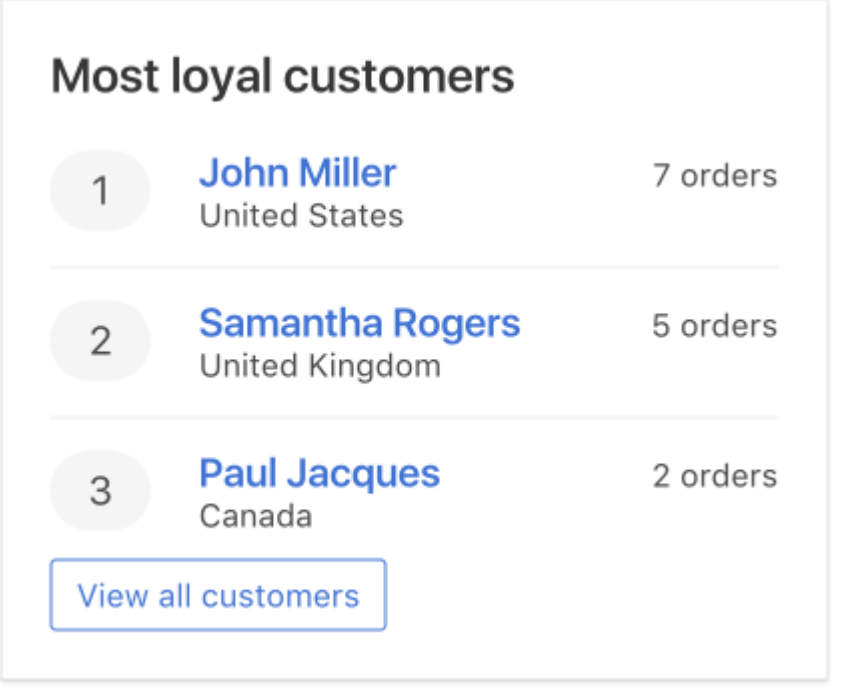
```
      <div class="media">
        <div class="media-left is-marginless">
          <p class="number">1</p>
        </div>
        <div class="media-content">
          <p class="title is-5 is-spaced is-marginless">
            <a href="edit-customer.html">John Miller</a>
          </p>
          <p class="subtitle is-6">
            United States
          </p>
        </div>
        <div class="media-right">
          7 orders
        </div>
      </div>

      <a class="button is-link is-outlined" href="customers.html">View all
customers</a>
    </div>
  </div>
</div>
```

- `button`: Bulma component. Adds base styles for buttons.
- `is-link`: Modifier class for buttons. Much like `is-primary`. Defaults to a blue color.
- `is-outlined`: Removed the background color of the button. Adds a colored border and colored text based on the other modifier.

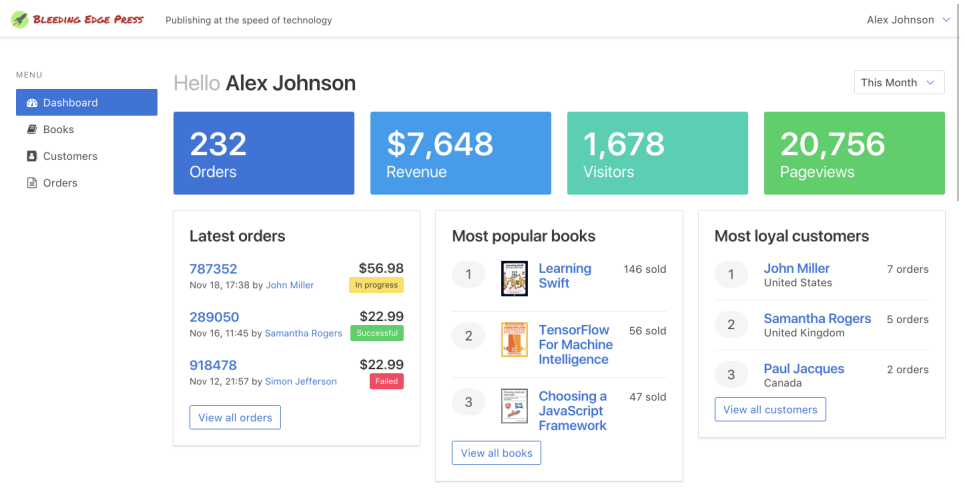


The Bulma `media` component is versatile enough to be re-used here, but with different data, and with one fewer `media-left`.

Add the second and third most loyal customers to the list:



The dashboard is now complete! Play with the content, change the modifier clases, and add more columns.



Summary

As you have seen in the building of this admin area, Bulma components come in various forms:

- Layout utilities (`section`, `columns`, `level`...)
- Single elements (box, `button`, `input`, `notification`...)
- Multipart components (`navbar`, `card`, `media`, `menu`, `pagination`...)
- Helper classes (`has-text-grey-light`, `is-hidden-tablet-only`...)

Most Bulma users like to combine all of these parts in a plethora of different ways, to build the UI their website needs. But most importantly, they like to *customize* their Bulma setup by providing their own colors and modifying the initial variables.

In the next chapter we will focus on using Bulma with Vanilla JavaScript.

9 Using Bulma with Vanilla JavaScript

Bulma does not come with any JavaScript implementation out of the box. In this chapter, however, you will learn how to control different components of the admin template using Vanilla JavaScript. If you've been following along, this chapter will be covering the following components that are used in the admin template:

- `Report a Bug Modal`
- `Mobile Menu Toggle`
- `Notifications`
- `Dropdowns`
- `Delete a book`
- `Delete a customer`

Report a Bug - Modal

In order to create the **Report a Bug** modal, you need the following components:

- A `Button` element
- Bulma's `Modal` component
- Bulma's `Notification` element

Note: To see the full code of the example used in this book take a look at the **book's accompanying GitHub page**.

For our example you need a `button` with `data-target` pointing to the `id` of the modal. In Bulma, modals can be shown and hidden using the `is-active` class. Using Vanilla Javascript you can add and remove the `is-active` class using the `classList` property, targeting the unique modal component `id` on the `button` click.

```
<!-- trigger button markup -->
<button class="button is-white open-modal-button" data-target="report-a-bug">
    <span class="icon">
        <i class="fa fa-bug"></i>
```

```
        </span>
        <span>
            Report a bug
        </span>
    </button>
```

You can also add a `notification` element to show success/error notifications on submission. Here is the markup of the `notifiction` element you will be using inside the `Modal`.

```
    <!-- Notification Element -->
    <div class="notification is-success is-hidden modal-success-notification">
        <span class="fa fa-bug"></span> Thank You. Your bug has been reported.
    </div>
```

In order to `close` the modal you need to add a `button` with the `delete` class, which will become a cross icon. In order to close the `Modal` via JavaScript you have to give the button `close-modal-button class`, which you will use to close the Modal through the JavaScript code. Here is the HTML markup for the close button.

```
    <button class="delete close-modal-button" aria-label="close"></button>
```

Lets combine all of the different pieces of the `Modal` component.

```
    <!-- modal markup -->
    <div class="modal" id="report-a-bug">
        <div class="modal-background"></div>
        <div class="modal-card">
            <header class="modal-card-head">
                <p class="modal-card-title">Report a Bug</p>
                <!-- Close Button -->
                <button class="delete close-modal-button" aria-label="close"></
    button>
            </header>
            <section class="modal-card-body">
            <!-- Notification Element -->
                <div class="notification is-success is-hidden modal-success-
    notification">
                    <span class="fa fa-bug"></span> Thank You. Your bug has been
    reported.
                </div>
                <textarea class="textarea" placeholder="Let us know what prob-
    lems you faced.">
                </textarea>
            </section>
            <footer class="modal-card-foot">
                <button class="button is-success send-bug-report">Send</button>
```

```
            <button class="button close-modal-button">Cancel</button>
        </footer>
    </div>
</div>
```

The JavaScript code for **Report a bug** modal is next. This is not just for a single modal, but it will take care of all modals you want to create throughout the application. You can also add a notification using Bulma's notification component and show/hide according to your requirements.

```
// Getting all the modals, close and trigger buttons
var modals = document.querySelectorAll('.modal');
var modalButtons = document.querySelectorAll('.open-modal-button');
var modalClose = document.querySelectorAll('.close-modal-button');

// For Success Message Notification
var successMessages = document.querySelectorAll('.modal-success-
notification');

// Adding a event listener to all the trigger buttons
if (modalButtons.length > 0) {
    modalButtons.forEach(button => {
        button.addEventListener('click', function() {
            document.getElementById(this.dataset.target).classList.add('is-
active');
        });
    });
}

// Adding event listeners to all the close buttons
if (modalClose.length > 0) {
    modalClose.forEach(closeButton => {
        closeButton.addEventListener('click', function() {
            modals.forEach(modal => {
                modal.classList.remove('is-active');
                // hiding success notification on closing the modal
                successMessages.forEach(message => {
                    message.classList.add('is-hidden');
                });
            });
        });
    });
}

// For Showing the Success Notification
var sendBugReport = document.querySelector('.send-bug-report');
if (sendBugReport !== null) {
    sendBugReport.addEventListener('click', function() {
        successMessages.forEach(message => {
```

```
            message.classList.remove('is-hidden');
        });
    });
}
```

Now let's explain the JavaScript code. For the report a bug modal, we are adding `eventListener` to the trigger button, which has a `class` of `open-modal-button`. Once the `Modal` is open we are then closing the `Modal` using the `close-modal-button` class.

Mobile menu toggle

Bulma changes the `navbar` into a mobile menu with a burger icon at a specific breakpoint. In order to make it workable you need to add some JavaScript code. You will create an event listener and toggle the `is-active` class. You also have to toggle the burger element's class to `is-active` to change the burger icon to close the icon.

```
var burger = document.querySelector('.burger');
var menu = document.querySelector('.navbar-menu')
if (burger !== null) {
    burger.addEventListener('click', function() {
        burger.classList.toggle('is-active');
        menu.classList.toggle('is-active');

    });
}
`
```

Notifications

Notifications can be used in many places to give users some extra information about the operations they perform. For example, you can have a notification added to the `Report a Bug` modal.

```
<div class="notification is-success is-hidden modal-success-notification">
    <button class="delete close-notification"></button>
        <span class="fa fa-bug"></span> Thanks. Your bug has been reported.
</div>
```

The notification shows up when you click send. So, in order to close the notification, you can click on the close icon inside the notification. To remove the notification you can create an event listener on the close icon using the `.close-notifiction` class and remove the notification. Here is the code you will need to add the functionality:

```
var closeNotification = document.querySelectorAll('.close-notification');
if (closeNotification.length > 0) {
    closeNotification.forEach(closeIcon => {
        closeIcon.addEventListener('click', () => {
            closeIcon.closest('.notification').remove();
        });
    });
}
```

Dropdowns

You can have both hoverable dropdowns and clickable dropdowns using Bulma.To make any menu open on hover, you have to add the `is-hoverable` class to the toggle element.

```
<div class="dropdown is-hoverable">
  <div class="dropdown-trigger">
    <button class="button" aria-haspopup="true" aria-controls="dropdown-
menu">
      <span>Hover me</span>
      <span class="icon is-small">
        <i class="fa fa-angle-down" aria-hidden="true"></i>
      </span>
    </button>
  </div>
  <div class="dropdown-menu" id="dropdown-menu" role="menu">
    <div class="dropdown-content">
      <div class="dropdown-item">
        <p>You can insert <strong>any type of content</strong> within the
dropdown menu.</p>
      </div>
    </div>
  </div>
</div>
```

You can also change the functionality from hover to click. To use the click for `dropdown` you need to create an event listener on the button and toggle the `is-active` class. The code next will make every dropdown item without the `is-hoverable` class active on click.

```
var dropdowns = document.querySelectorAll('.dropdown:not(.is-hoverable)');
if (dropdowns.length > 0) {
    dropdowns.forEach(dropdown => {
        dropdown.addEventListener('click', event => {
            event.stopPropagation();
            dropdown.classList.toggle('is-active');
        });
    });
```

```
    document.addEventListener('click', event => {
        dropdowns.forEach(dropdown => {
            dropdown.classList.remove('is-active');
        });
    });
}
```

Delete a book item from books page

You can also delete a book item from the book's list on the book's page. Here is the code you will need to achieve it. You need to create an event listener on the delete button with each book and remove the closest column.

```
// for delete an item
var deleteItem = document.querySelectorAll('.delete-item');
if (deleteItem.length > 0) {
    deleteItem.forEach(button => {
        button.addEventListener('click', function() {
            button.closest('.column').remove();
        });
    })
}
```

Delete a customer from customer page

To delete a customer from the customer list on the customer's page you can use the code below. You have to add an event listener on the delete button, and remove the closest row.

```
//for deleting a customer
var deleteUserButton = document.querySelectorAll('.delete-user');
if (deleteUserButton.length > 0) {
    deleteUserButton.forEach(button => {
        button.addEventListener('click', function() {
            button.closest('tr').remove();

        });
    });
}
```

Summary

You should now have a basic understanding about how to control different components of the Admin Template in Bulma using Vanilla JavaScript.

In the next chapter we use Bulma with Angular.

10 Using Bulma with Angular

As you know, **Angular** is a platform that makes it easy to build applications with the web. Angular combines declarative templates, dependency injection, end-to-end tooling, and integrated best practices to solve developmental challenges. But it does *not* provide you with a rich UI experience. This is where Bulma comes in.

As illustrated in examples of using Bulma with JavaScript, it's really easy right? Now let's integrate our Bulma templates with the Angular framework! So, what do you need?

- Knowledge of CLI
- Node.js
- Angular CLI

If you don't have these installed, it's easy to get up and running with Angular. You simply just need to download Node.js from the official website and follow the installation instructions. After Node is installed, you need to install the Angular CLI via NPM.

```
Node.js Website: https://nodejs.org/en/

# Install Angular CLI
npm install -g @angular/cli
```

Project preparation

Now let's create a brand new Bulma and Angular project with those step-by-step instructions. This chapter is going to rely on the command line pretty heavily. Don't worry though, this chapter will document the commands needed to get your project up and running.

- Navigate to your project directory and create a folder.

```
mkdir my-repos
cd my-repos
```

- Creating a new application is not too complicated, but if you want to learn more features, you can read about them at `cli.angular.io`. Angular CLI is going to install of the dependencies needed to get the local environment up and running.

```
ng new sample-application --style scss --routing
cd sample-application
```

- Add **Bulma** to your Angular application.

```
npm install bulma --save
npm install font-awesome --save
```

Note: This project is also using Font Awesome. Make sure you check out their documentation for more information.

- Let's edit the `.angular-cli.json` file by adding Bulma and Font Awesome to the styles section.

```
../node_modules/bulma/bulma.sass
../node_modules/font-awesome/scss/font-awesome.scss
```

The styles section of your `.angular-cli.json` file should resemble this:

```
"styles": [
  "../node_modules/bulma/bulma.sass",
  "../node_modules/font-awesome/scss/font-awesome.scss",
  "styles.scss"
],
```

- You're almost there. Let's start the application by running:

```
npm start
```

or

```
ng serve --open
```

Remember, you can customize these commands later on in the `package.json` file.

Application

The application that you will be building is a simple book store for a book publishing company. Let's begin with a dashboard, and then add the books' catalog, customers, and orders lists. Is it possible to achieve all of this with Bulma? Sure, you have everything that is needed for such functionality!

All you need is to generate the components.

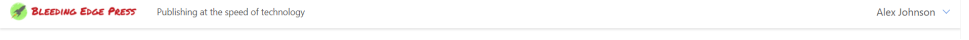
```
ng g component components/[component-name]
```

Please remember that this is a common way to create any component. You will most likely use this same console command frequently in the future.

Components

Now you can open the `app.component.html` and fill it in with **html** markup. This component will be responsible for navigation throughout the application.

First, let's create a top menu. It will look like this:



What will you need? `navbar`, `navbar-brand`, `navbar-item`, `navbar-start`, and `navbar-end` classes will help you do the job. By combining these classes you should have this markup, or something similar to this:

```
<nav class="navbar has-shadow">
    <div class="navbar-brand">
        <a class="navbar-item">
            <img src="assets/images/logo.png">
        </a>
        <div [ngClass]="{'is-active': active==true}" class="navbar-burger
burger" (click)="active=!active">
            <span></span>
            <span></span>
            <span></span>
        </div>
    </div>

    <div [ngClass]="{'is-active': active==true}" class="navbar-menu">
        <div class="navbar-start">
            <div class="navbar-item">
                <small>Publishing at the speed of technology</small>
            </div>
        </div>

        <div class="navbar-end">
            <div class="navbar-item has-dropdown is-hoverable">
                <div class="navbar-link">
                    Alex Johnson
                </div>
                <div class="navbar-dropdown">
```

```
                    <a class="navbar-item" (click)="action()">
                        <div>
                            <span class="icon is-small">
                  <i class="fa fa-user-circle-o"></i>
                </span> Profile
                        </div>
                    </a>
                    <a class="navbar-item" (click)="action()">
                        <div>
                            <span class="icon is-small">
                  <i class="fa fa-bug"></i>
                </span> Report bug
                        </div>
                    </a>
                    <a class="navbar-item" (click)="action()">
                        <div>
                            <span class="icon is-small">
                  <i class="fa fa-sign-out"></i>
                </span> Sign Out
                        </div>
                    </a>
                </div>
            </div>
        </div>
    </div>
</nav>
```

As for the sidebar, it will simply need the `menu`, `menu-list`, and `menu-label` classes.

```
<section class="section ">
    <div class="columns ">
        <div class="column is-4-tablet is-3-desktop is-2-widescreen">
            <nav class="menu">
                <p class="menu-label">
                    Menu
                </p>
                <ul class="menu-list">
                    <li>
                        <a [routerLinkActive]="['is-active']" [router-
Link]="['/dashboard']">
                            <span class="icon">
                              <i class="fa fa-tachometer"></i>
                            </span> Dashboard
                        </a>
                    </li>
                    <li>
                        <a [routerLinkActive]="['is-active']" [router-
Link]="['/books']">
                            <span class="icon">
```

```
                            <i class="fa fa-book"></i>
                        </span> Books
                    </a>
                </li>
                <li>
                    <a [routerLinkActive]="['is-active']" [router-
Link]="['/customers']">
                        <span class="icon">
                            <i class="fa fa-address-book"></i>
                        </span> Customers
                    </a>
                </li>
                <li>
                    <a [routerLinkActive]="['is-active']" [router-
Link]="['/orders']">
                        <span class="icon">
                            <i class="fa fa-file-text-o"></i>
                        </span> Orders
                    </a>
                </li>
            </ul>
        </nav>
    </div>
    <main class="column ">
        <router-outlet></router-outlet>
    </main>
</div>
</section>
```

You should replace the content container with `<router-outlet></router-outlet>`. It should look something like this:

```
<main class="column ">
    <router-outlet></router-outlet>
</main>
```

Now let's add a child component to the application. You can do it manually or by running a command. Angular CLI can generate a component for you.

```
ng g component components/dashboard -m routing.module
```

After the component gets generated, go ahead and open the `dashboard.component.html` and write some custom markup. For the sake of this chapter, the book is going to use the example above. Remember that you are inserting *only* the content part of the markup.

To see the full code of the example used in this book see the **books accompanying GitHub page**.

- Content Header


```
<div class="level">
    <div class="level-left">
        <h1 class="subtitle is-3">
            <span class="has-text-grey-light">Hello</span>
            <strong>Alex Johnson</strong>
        </h1>
    </div>
    <div class="level-right">
        <div class="select">
            <select [(ngModel)]="filter" (ngModelChange)="onChange($event)">
                <option>Today</option>
                <option>Yesterday</option>
                <option>This Week</option>
                <option>This Month</option>
                <option>This Year</option>
                <option>All time</option>
            </select>
        </div>
    </div>
</div>
```

- The Summary Tiles

```
    <div class="column is-12-tablet is-6-desktop is-3-widescreen">
        <div class="notification is-link has-text">
            <p class="title is-1">{{statistics[0].orders}}</p>
            <p class="subtitle is-4">Orders</p>
        </div>
    </div>

    <div class="column is-12-tablet is-6-desktop is-3-widescreen">
        <div class="notification is-info has-text">
            <p class="title is-1">${{statistics[0].revenue}}</p>
            <p class="subtitle is-4">Revenue</p>
        </div>
    </div>

    <div class="column is-12-tablet is-6-desktop is-3-widescreen">
        <div class="notification is-primary has-text">
            <p class="title is-1">{{statistics[0].visitors}}</p>
            <p class="subtitle is-4">Visitors</p>
        </div>
    </div>
```

```
    <div class="column is-12-tablet is-6-desktop is-3-widescreen">
        <div class="notification is-success has-text">
            <p class="title is-1">{{statistics[0].pageviews}}</p>
            <p class="subtitle is-4">Pageviews</p>
        </div>
    </div>
```

- Content Cards

```
    <div class="column is-12-tablet is-6-desktop is-4-fullhd">
        <div class="card">
            <div class="card-content">
                <h2 class="title is-4">
                    Latest orders
                </h2>

                <div class="level" *ngFor="let order of orders; let i =
index">
                    <div class="level-left">
                        <div>
                            <p class="title is-5 is-marginless">
                                <a [routerLink]="['/orders-edit']" [queryPar-
ams]="{id: order.id }">{{ order.number }}</a>
                            </p>
                            <small>
                                {{ order.date }} by
                                {{ order.customer }}
                            </small>
                        </div>
                    </div>
                    <div class="level-right">
                        <div class="has-text-right">
                            <p class="title is-5 is-marginless">
                                ${{ order.total }}
                            </p>
                            <span *ngIf="order.status === 'In progress'"
class="tag is-warning">{{ order.status }}</span>
                            <span *ngIf="order.status === 'Successful'"
class="tag is-success">{{ order.status }}</span>
                            <span *ngIf="order.status === 'Failed'"
class="tag is-failed">{{ order.status }}</span>
                        </div>
                    </div>
                </div>

                <a class="button is-link is-outlined" [routerLink]="['/or-
ders']">View all orders</a>
            </div>
        </div>
```

```
        </div>

        <div class="column is-12-tablet is-6-desktop is-4-fullhd">
            <div class="card">
                <div class="card-content">
                    <h2 class="title is-4">
                        Most popular books
                    </h2>

                    <div class="media" *ngFor="let book of books; let i = index">
                        <div class="media-left is-marginless">
                            <p class="number">{{i + 1}}</p>
                        </div>
                        <div class="media-left">
                            <img src="assets/images/{{book.image}}" width="40">
                        </div>
                        <div class="media-content">
                            <p class="title is-5 is-spaced is-marginless">
                                <a [routerLink]="['/books-edit']" [queryPar-
ams]="{id: book.id }">{{book.title}}</a>
                            </p>
                        </div>
                        <div class="media-right">
                            {{ filter }}
                        </div>
                    </div>

                    <a class="button is-link is-outlined" [routerLink]="['/
books']">View all books</a>
                </div>
            </div>
        </div>

        <div class="column is-12-tablet is-6-desktop is-4-fullhd">
            <div class="card">
                <div class="card-content">
                    <h2 class="title is-4">
                        Most loyal customers
                    </h2>

                    <div class="media" *ngFor="let customer of customers; let i
= index">
                        <div class="media-left is-marginless">
                            <p class="number">{{i + 1}}</p>
                        </div>
                        <div class="media-content">
                            <p class="title is-5 is-spaced is-marginless">
                                <a [routerLink]="['/customers-edit']" [queryPar-
ams]="{id: customer.id }">{{ customer.name }}</a>
                            </p>
```

```
                        <p class="subtitle is-6">
                            <td>{{ customer.country }}</td>

                        </p>
                    </div>
                    <div class="media-right">
                        {{ customer.orders }} orders
                    </div>
                </div>

                <a class="button is-link is-outlined" [routerLink]="['/cus-
tomers']">View all customers</a>
            </div>
        </div>
    </div>
```

Let's add an `Orders` component to the application. Again, you can create a component manually or by running a command.

```
ng g component components/orders -m routing.module
```

Now you can open the `orders.component.html` and fill it in with some custom HTML markup. There will be three main parts.

1. Header


```
<h1 class="title ">Orders</h1>

<nav class="level">
    <div class="level-left">
        <div class="level-item">
            <p class="subtitle is-5">
                <strong>2</strong> orders
            </p>
        </div>
        <div class="level-item is-hidden-tablet-only">
            <div class="field has-addons">
                <p class="control">
                    <input class="input" type="text" placeholder="Order #â
€¦" [(ngModel)]="userFilter.number">
                </p>
                <p class="control">
                    <button class="button" (click)="userFilter.number = ''">
```

```
                Clear
              </button>
                </p>
            </div>
        </div>
    </div>

    <div class="level-right">
        <p class="level-item" (click)="userFilter.status =
''"><a><strong>All</strong></a></p>
        <p class="level-item" (click)="userFilter.status = 'In pro-
gress'"><a>In progress</a></p>
        <p class="level-item" (click)="userFilter.status = 'Success-
ful'"><a>Successful</a></p>
        <p class="level-item" (click)="userFilter.status =
'Failed'"><a>Failed</a></p>
    </div>
</nav>
```

1. Grid

```
<table class="table is-hoverable is-fullwidth">
    <thead>
        <tr>
            <th>Order #</th>
            <th>Customer</th>
            <th>Date</th>
            <th>Books</th>
            <th>Status</th>
            <th class="has-text-right">Total</th>
        </tr>
    </thead>
    <tfoot>
        <tr>
            <th>Order #</th>
            <th>Customer</th>
            <th>Date</th>
            <th>Books</th>
            <th>Status</th>
            <th class="has-text-right">Total</th>
        </tr>
    </tfoot>
    <tbody>
        <tr *ngFor="let order of orders | filterBy: userFilter | orderBy:
order">
            <td>
                <a [routerLink]="['/orders-edit']" [queryParams]="{id: or-
der.id }"><strong>{{ order.number }}</strong></a>
            </td>
            <td>
```

```
                <a [routerLink]="['/customers']">{{ order.customer }}</a>
            </td>
            <td>{{ order.date }}</td>
            <td>{{ order.books }}</td>
            <td>
                <span *ngIf="order.status === 'In progress'" class="tag is-
warning">{{ order.status }}</span>
                <span *ngIf="order.status === 'Successful'" class="tag is-
success">{{ order.status }}</span>
            </td>
            <td class="has-text-right">${{ order.total }}</td>
        </tr>
    </tbody>
</table>
```

1. Pagination

```
<nav class="pagination">
    <a class="pagination-previous">Previous</a>
    <a class="pagination-next">Next page</a>
    <ul class="pagination-list">
        <li>
            <a class="pagination-link">1</a>
        </li>
        <li>
            <span class="pagination-ellipsis">â€¦</span>
        </li>
        <li>
            <a class="pagination-link">1</a>
        </li>
    </ul>
</nav>
```

Let's add a `customers` component to our application.

```
ng g component components/customers -m routing.module
```

Now you can open the `customers.component.html` and fill it in with some more custom HTML. It will be similar to the previous components.

1. Header

```
<h1 class="title ">Customers</h1>

<nav class="level">
    <div class="level-left">
        <div class="level-item">
            <p class="subtitle is-5">
                <strong>{{ (customers | filterBy: userFilter).length }}</
strong> customers
            </p>
        </div>

        <p class="level-item">
            <a class="button is-success" (click)="add()">New</a>
        </p>

        <div class="level-item is-hidden-tablet-only">
            <div class="field has-addons">
                <p class="control">
                    <input class="input" type="text" placeholder="Nameâ€¦"
[(ngModel)]="userFilter.name">
                </p>
                <p class="control">
                    <button class="button" (click)="userFilter.name = ''">
                        Clear
                    </button>
                </p>
            </div>
        </div>
    </div>

    <div class="level-right">
        <p class="level-item" (click)="userFilter.hasOrders = ''">
            <a>
                <strong>All</strong>
            </a>
        </p>
        <p class="level-item" (click)="userFilter.hasOrders = true">
            <a>With orders</a>
        </p>
        <p class="level-item" (click)="userFilter.hasOrders = false">
            <a>Without orders</a>
        </p>
    </div>
</nav>
```

1. Grid

```
<table class="table is-hoverable is-fullwidth">
    <thead>
        <tr>
```

```
            <th class="is-narrow">
                <input type="checkbox">
            </th>
            <th>Name</th>
            <th>Email</th>
            <th>Country</th>
            <th>Orders</th>
            <th>Actions</th>
        </tr>
    </thead>
    <tfoot>
        <tr>
            <th class="is-narrow">
                <input type="checkbox">
            </th>
            <th>Name</th>
            <th>Email</th>
            <th>Country</th>
            <th>Orders</th>
            <th>Actions</th>
        </tr>
    </tfoot>
    <tbody>
        <tr *ngFor="let customer of customers | filterBy: userFilter | order-
By: order">
            <td>
                <input type="checkbox">
            </td>
            <td>
                <a [routerLink]="['/customers-edit']" [queryParams]="{id:
customer.id }">
                    <strong>{{ customer.name }}</strong>
                </a>
            </td>
            <td>
                <code>{{ customer.email }}</code>
            </td>
            <td>{{ customer.country }}</td>
            <td>
                <a [routerLink]="['/orders']">{{ customer.orders }}</a>
            </td>
            <td>
                <div class="buttons">
                    <a class="button is-small" [routerLink]="['/customers-
edit']" [queryParams]="{id: customer.id }">Edit</a>
                    <a class="button is-small" (click)="delete()">Delete</a>
                </div>
            </td>
        </tr>
```

```
    </tbody>
</table>
```

1. Pagination

```
<nav class="pagination">
    <a class="pagination-previous">Previous</a>
    <a class="pagination-next">Next page</a>
    <ul class="pagination-list">
        <li>
            <a class="pagination-link">1</a>
        </li>
        <li>
            <span class="pagination-ellipsis">â€¦</span>
        </li>
        <li>
            <a class="pagination-link">1</a>
        </li>
    </ul>
</nav>
```

Continuing on, let's create a books component to your application.

```
ng g component components/books -m routing.module
```

Now you can open the `books.component.html` and fill it in with HTML markup that you want. Remember that this code snippet below is *only* the content part of the markup.

1. Header

```
<h1 class="title ">Books</h1>

<nav class="level">
    <div class="level-left">
        <div class="level-item">
            <p class="subtitle is-5">
                <strong>{{ (books | filterBy: userFilter).length }}</strong>
books
            </p>
        </div>

        <p class="level-item">
            <a class="button is-success" (click)="add()">New</a>
        </p>

        <div class="level-item is-hidden-tablet-only">
            <div class="field has-addons">
                <p class="control">
                    <input class="input" type="text" placeholder="Book ti-
```

```
tle..." [(ngModel)]="userFilter.title">
                </p>
                <p class="control">
                    <button class="button" (click)="userFilter.title = ''">
                    Clear
                  </button>
                </p>
            </div>
        </div>
    </div>


    <div class="level-right">
        <div class="level-item">
            Order by
        </div>
        <div class="level-item">
            <div class="select">
                <select [(ngModel)]="order">
                  <option value="title">Title</option>
                  <option value="price">Price</option>
                  <option value="pages">Page count</option>
                </select>
            </div>
        </div>
    </div>
</nav>
```

1. Tiles

```
<div class="columns is-multiline">
    <div class="column is-12-tablet is-6-desktop is-4-widescreen"
*ngFor="let book of books | filterBy: userFilter | orderBy: order">
        <article class="box">
            <div class="media">
                <aside class="media-left">
                    <img src="assets/images/{{book.image}}" width="80">
                </aside>
                <div class="media-content">
                    <p class="title is-5 is-spaced is-marginless">
                        <a [routerLink]="['/books-edit']" [queryPar-
ams]="{id: book.id }">{{book.title}}</a>
                    </p>
                    <p class="subtitle is-marginless">
                        ${{book.price}}
                    </p>
                    <div class="content is-small">
                        {{book.pages}} pages
                        <br> ISBN: {{book.ISBN}}
                        <br>
```

```
                        <a [routerLink]="['/books-edit']" [queryPar-
ams]="{id: book.id }">Edit</a>
                        <span>Â·</span>
                        <a>Delete</a>
                        <p></p>
                    </div>
                </div>
            </div>
        </article>
    </div>
</div>
```

1. Pagination

```
<nav class="pagination">
    <a class="pagination-previous">Previous</a>
    <a class="pagination-next">Next page</a>
    <ul class="pagination-list">
        <li>
            <a class="pagination-link">1</a>
        </li>
        <li>
            <span class="pagination-ellipsis">â€¦</span>
        </li>
        <li>
            <a class="pagination-link">1</a>
        </li>
    </ul>
</nav>
```

Summary

Now you can run the application. As you can see it is really easy to use the Bulma framework with Angular! I hope you understand *why* you use Bulma classes with Angular instead of *how* you use them.

The next chapter covers using Bulma with VueJS.

Using Bulma with VueJS 11

In this chapter we implement parts of the admin dashboard from earlier with the progressive JavaScript framework VueJS. It's important to keep in mind that this is *not* a tutorial on VueJS itself. Rather, it's more about implementing Bulma with your VueJS application.

If you need more help with VueJS head over to the **VueJS official documentation**, which like Bulma's documentation is *very* good and is an easy read as far as documentation is concerned.

Installing Vue-CLI

In this chapter, you will be using Vue's command line tool, `vue-cli`. To get started with `vue-cli`, run the following commands:

```
npm install -g vue-cli
vue init <template> <project-name>
cd <project-name>
npm install
npm run dev
```

This chapter will be using the *webpack-simple* template. This is one of many templates that you can choose to download when creating your Vue application with `vue-cli`. Make sure you replace `<template>` with `webpack-single` during the `vue-cli` setup. This chapter is also going to make use of **Vue-Router** to easily handle navigating between "pages" on the dashboard. Routing is essential for every single page application. With it, you can mount a single parent component with its child components based on a URL.

Note: There are a total of *six* different templates to choose from with the CLI. To find out what they include check out **the Vue CLI github repo**.

Before you jump into setting everything up, there are some prerequisites for this chapter. In order to have a basic understanding of integrating Bulma with Vue, you need the following installed:

- **Node**

- **NPM**
- **Vue CLI**

Setting up the Vue project

Let's start by installing `vue-cli` with a fresh VueJS project. As mentioned earlier, this chapter uses the **webpack-simple** template with "bulma-dashboard" as the name of the project.

The directory structure should look similiar to this:

- bulma-dashboard [project name folder]
    - node_modules/
    - src/
        - assets/
        - App.vue
        - main.js
    - index.html
    - package.json
    - README.md
    - webpack.config.js

Preparing pages

Before continuing with implementing `vue-router` you should set up skeletons for all of your components. Create a new `pages/` directory inside the `src/` folder. Next, create `.vue` files for the components: `Dashboard.vue`, `Books.vue`, `Orders.vue`, and `Login.vue`. Your text editor of choice might be able to create `.vue` files already, but if not, here's a little snippet for what the every `.vue` file should include:

```
<template>

</template>

<script>
  export default {
    name: [ INSERT NAME OF COMPONENT ]
  }
</script>

<style>

</style>
```

Note: If you have a Sass loader installed and configured with Webpack, add the `lang="sass"` attribute to your `style` tag.

Switch out **[INSERT NAME OF COMPONENT]** for your page name. For example, "Books."

Vue-Router

Now, add `vue-router` to the project. There are a few different ways of doing this. Here is one way of installing it while keeping the code organized.

- Install `vue-router`:

```
npm install vue-router
```

- Create a folder named `router/` to the root folder.
- Create a file named `index.js` inside this new `router/` folder.
- Inside the file, import Vue-Router with components you want to route to.

```
import VueRouter from 'vue-router'
```

- Feed the `routes:{}` object to the `new Router()`. Your `routes` object should contain an array of components and their names.
- Lastly, inside your `main.js` file, import the new router `index.js` file and add it when initializing Vue, inside the `new vue()` instance.

Your `router/index.js` file should resemble something close to this:

```
import Vue from "vue";
import Router from "vue-router";
import Dashboard from "../pages/Dashboard.vue";
import /*...(rest of pages) */

Vue.use(Router);

export default new Router({
  routes: [
    {
      path: "/",
      redirect: '/dashboard'
    },
    {
      path: "/dashboard",
      name: "Dashboard",
      component: Dashboard,
    },
    /*...(rest of pages) */
```

```
    ],
    linkActiveClass: 'is-active'  /* change to Bulma's active nav link */
});
```

The `main.js` file should resemble something like the following:

```
import Vue from "vue";
import App from "./App.vue";
import router from "./router";

/* other stuff */

new Vue({
  el: "#app",
  router,
  render: h => h(App),
});
```

That is it for the simple router, but if you wish to learn more check out the **Vue-Router docs**.

You should now be able to run the application with the following command:

```
npm run dev
```

Installing Bulma

To round off this setup section, let's add the latest version of **Bulmas CSS** to the Vue project. There are two main ways that this can be done: Adding it via a CDN with a `<link>` tag or adding it via NPM.

Option 1: Adding Bulma via a CDN

In case you are only testing out Bulma and you know you won't need any customization, adding it via a `<link>` tag might suffice. In that case, open the `index.html` file inside your project root, and inside the `<head>` tag add Bulma via a CDN just like any other stylesheet in a website.

```
<link href="https://cdnjs.cloudflare.com/ajax/libs/bulma/0.6.2/css/
bulma.min.css" rel="stylesheet">
```

Option 2: Adding Bulma via NPM (Recommended)

This is the recommended way of adding external libraries in single page applications. When creating your project with `vue-cli` you are also installing Webpack with configurations already made. Adding Bulma via NPM will add the CSS framework and will bundle it in your `build.js`

INSTALL BULMA THROUGH NPM

```
npm install bulma --save
```

Then open up `main.js` and from here, you import Bulma.

```
import './../node_modules/bulma/css/bulma.css';
```

This is easy and simple, with one caveat. In order to customize any Bulma variables unique to your application, you need to create a `styles.css` file inside your `src/assets/` directory. From here, you can start importing the *initial variables* and *function* files. Then add your customizations, and finally, import the main bulma file.

```
@import '../../node_modules/bulma/sass/utilities/initial-variables';
@import '../../node_modules/bulma/sass/utilities/functions';

$primary: #ffb3b3; /* changes primary color to pink */

@import '../../node_modules/bulma/bulma';
```

Then change the import in the `main.js` file to the custom styles file instead.

```
import router from "./router";

import './assets/custom.scss';
```

BONUS: CREATING AN ALIAS FOR YOUR BULMA DEPENDENCY

As stated before, if you import Bulma with NPM, one way to use it is with an ES6 import statement. However, this path needs to be a *relative* link. You can easily make this absolute with a Webpack alias.

To create an alias in Webpack, open up your `build/webpack.dev.conf.js` file and paste the following code above the *module* object.

```
resolve: {
    extensions: ['.css'],
    alias: {
      'bulma': resolve('node_modules/bulma/css/bulma.css'),
```

```
    }
}
```

This will create the alias. From here you can now import Bulma with an absolute link that is a little easier to read.

```
import 'bulma';
```

Note: As with everything in JavaScript, there are several Vue+Bulma packages around the web to install; all with their own pros and cons.

Make use of Font-Awesome

Finally, you'll want to use Font-Awesome fonts in the app, so for this you can simply link to Font-Awesome from a CDN.

Open your `index.html` file and add the following to the `<head>`section.

```
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/4.7.0/css/font-awesome.min.css">
```

The final folder structure looks something like this:

- bulma-dashboard [main project folder]
    - node_modules/
    - src/
        - assets/
            - images/
            - styles.scss
            - logo.png
        - pages/
            - Books.vue
            - Customers.vue
            - Dashboard.vue
            - Login.vue
            - Orders.vue
        - router
            - index.js
        - App.vue
        - main.js
    - index.html
    - package.json

- README.md
- webpack.config.js

In the next section, you should pull in the base skeleton and templates for the app in the earlier HTML section.

Setting up components with Vue

Note: Depending on your Vue knowledge and whether or not you are following along with the previous examples, you can skip to the next section where this chapter will explain implementing more Bulma functionality. The snippets from now on might not be complete, but for full code see the **books accompanying GitHub page**.

In the last chapter you setup Vue with routing and installed Bulma. Now it's time to move over the HTML from previous chapters into the `.vue` files. You will start with setting the main template in the `App.vue` file and then create some of the components from the previous chapters inside the `pages` folder. Later, you will finish off a few components with more interactive functionality.

Admin skeleton

Let's start with moving a little bit of code from `/html/dashboard.html` to the `App.vue` file. One thing to note is that we remove all "content" from the file because that code resides in each component's own `.vue` file.

```
<div id="app">
  <nav class="navbar has-shadow">
    <div class="navbar-brand">
      <a class="navbar-item" href="#">
        <img src="logo.png" alt="Bleeding Edge Press">
      </a>
      <div class="navbar-burger burger">
        <span></span>
        <span></span>
        <span></span>
      </div>
    </div>

    <div class="navbar-menu">
      <div class="navbar-start">
        <div class="navbar-item">
          <small>Publishing at the speed of technology</small>
        </div>
      </div>
```

```
        <div class="navbar-end">
          <div class="navbar-item has-dropdown is-hoverable">
            <div class="navbar-link">
              John Doe
            </div>
            <div class="navbar-dropdown">
              <a class="navbar-item">
                  <span class="icon is-small">
                    <i class="fa fa-user-circle-o"></i>
                  </span> Profile
              </a>
              <a class="navbar-item">
                  <span class="icon is-small">
                    <i class="fa fa-bug"></i>
                  </span> Report bug
              </a>
              <a class="navbar-item">
                  <span class="icon is-small">
                    <i class="fa fa-sign-out"></i>
                  </span> Sign Out
              </a>
            </div>
          </div>
        </div>
      </div>
    </nav>

    <section class="section">
      <div class="columns">
        <div class="column is-4-tablet is-3-desktop is-2-widescreen">
          <aside class="menu">
            <p class="menu-label">Menu</p>
            <ul class="menu-list">
              <li>
                <router-link to="/dashboard">
                  <span class="icon ">
                    <i class="fa fa-tachometer"></i>
                  </span>Dashboard</router-link>
              </li>
              <li>
                <router-link to="/books">
                  <span class="icon">
                    <i class="fa fa-book"></i>
                  </span> Books
                </router-link>
              </li>
              <li>
                <router-link to="/customers">
                  <span class="icon">
                    <i class="fa fa-address-book"></i>
```

```
              </span> Customers
            </router-link>
          </li>
          <li>
            <router-link to="/orders">
              <span class="icon">
                <i class="fa fa-file-text-o"></i>
              </span>
              Orders
            </router-link>
          </li>
        </ul>
      </aside>
    </div>
    <main class="column">
              <router-view></router-view>
          </main>
  </div>
 </section>
</div>
```

Let's take a closer look at the snippet above to see what's different compared to the pure HTML version. You might have noticed two things, especially `<router-link>` and `<router-view>`. These two tags are used because of `vue-router`, which you installed with Vue in the last section. As you might recall, this makes Vue able to handle routing. By visiting a route or URL, you can mount and render a certain component; this makes the app a *single page application*.

The `<router-link></router-link>` tag translates into a plain `<a href="#"></a>` tag. The `to=""` attribute corresponds to a specific **path** variable that you defined in your `/router/index.js` file (snippet below).

```
routes: [
  {
    path: "/dashboard",
    name: "Dashboard",
    component: Dashboard,
  }...
```

The `<router-view></router-view>` tag is where the contents of the current component (route) will be displayed. So after you have logged into the site, the **Dashboard** component will be shown and the code from `Dashboard.vue` will be inserted in the DOM where `<router-view></router-view>` is placed.

Implementing the dashboard

In your file structure you should have a `pages/` folder, and inside that folder you should have an empty `Dashboard.vue` file. In this file you will add the code from the main area from the HTML version.

Let's start with the top part of the Dashboard page, which contains the logged in users name and a dropdown used to filter results.

```
<div class="level">
    <div class="level-left">
        <h1 class="subtitle is-3">
            <span class="has-text-grey-light">Hello</span>
            <strong>Alex Johnson</strong>
        </h1>
    </div>
    <div class="level-right">
        <div class="select">
            <select @change="changeStats">
                <option value="today" selected>Today</option>
                <option value="yesterday">Yesterday</option>
                <option value="week">This Week</option>
                <option value="month">This Month</option>
                <option value="year">This Year</option>
                <option value="alltime">All time</option>
            </select>
        </div>
    </div>
</div>
```

There is nothing really different from the HTML version here, except the values on the `<option>` and a `@change="changeStats"` on the `<select>`. That is Vue code for listening on the *change event* of the select. When you select a different option, the `changeStats()` method gets fired and it changes the stats on display.

So, let's implement the stats section next, together with a data object, so that you can change the stats.

```
<div class="columns is-multiline">
  <div class="column is-12-tablet is-6-desktop is-3-widescreen">
    <div class="notification is-link has-text">
      <p class="title is-1">{{ selectedStats.orders }}</p>
      <p class="subtitle is-4">Orders</p>
    </div>
  </div>

  <div class="column is-12-tablet is-6-desktop is-3-widescreen">
    <div class="notification is-info has-text">
```

```
        <p class="title is-1">${{ selectedStats.revenue }}</p>
        <p class="subtitle is-4">Revenue</p>
      </div>
    </div>

    <div class="column is-12-tablet is-6-desktop is-3-widescreen">
      <div class="notification is-primary has-text">
        <p class="title is-1">{{ selectedStats.visitors }}</p>
        <p class="subtitle is-4">Visitors</p>
      </div>
    </div>

    <div class="column is-12-tablet is-6-desktop is-3-widescreen">
      <div class="notification is-success has-text">
        <p class="title is-1">{{ selectedStats.pageviews }}</p>
        <p class="subtitle is-4">Pageviews</p>
      </div>
    </div>
  </div>
```

Here you are introduced to Vue's template syntax `{{ selectedStats.revenue }}`, also known as string interpolation. The text inside the curly-brackets `{{ }}` are variables from your data object. Go ahead and add the following data object inside the `data() {}` method.

```
export default {
  name: 'Dashboard',
  data() {
    return {
      stats: {
        today: {
          orders: "232",
          revenue: "7,648",
          visitors: "1,678",
          pageviews: "20,756"
        },
        yesterday: {
          orders: "200",
          revenue: "5,465",
          visitors: "1,400",
          pageviews: "18,556"
        },
        week: {...},
        month: {...},
        allTime: {...}
      }
    }
  }
}
```

Now that you we have some data and code in place for the stats boxes, you can implement the `changeStats()` method. Below the `data()` method you can add the following piece of code, which also sets today's stats when the page loads.

```
mounted: function(){
  this.selectedStats = this.stats.today;
},
methods: {
  changeStats(event) {
    this.selectedStats = this.stats[event.target.value];
  }
}
```

Lastly, let's take a closer look at the first column of three with the latest orders. It contains a list of the latest orders with `ordernumber`, `date`, `customer`, `price`, and `status` of the order. The order status code contains Bulma's `.tag` class together with a modifier class, which gives the viewer a visual understanding of the order status.

Note: Modifier classes in Bulma begin with `is-` or `has-`.

This is what the template code for the *LatestOrders* list looks like:

```
<div class="column is-12-tablet is-6-desktop is-4-fullhd">
  <div class="card">
    <div class="card-content">
      <h2 class="title is-4">Latest orders</h2>

      <template v-for="(order, key) in orders">
        <div class="level" :key="order.id">
          <div class="level-left">
            <div>
              <p class="title is-5 is-marginless">
                <router-link to="/edit-order">{{ order.id }}</router-link>
              </p>
              <small>{{ order.date }} by <router-link to="/edit-
customer">{{ order.purchasedBy }}</router-link></small>
            </div>
          </div>
          <div class="level-right">
            <div class="has-text-right">
              <p class="title is-5 is-marginless">${{ order.price }}</p>
              <span class="tag" :class="order.status.class">{{ order.sta-
tus.label }}</span>
            </div>
          </div>
        </div>
      </template>
      <router-link class="button is-link is-outlined" to="/orders">View all
orders</router-link>
    </div>
```

```
    </div>
  </div>
```

So what is happening here is that Vue is looping through the `orders` array and for each `order`, Vue print's out its details. The thing to point out here is the `:class=""` attribute. This is a special Vue attribute that let's you manipulate the class list by binding to your data. In the snippet above we bound it to an orders status class value. What does this mean? Let's look at a shortened example of the data object for an order.

Note: A colon (`:`) is shorthand for `v-bind:`. So, the `:class=""` above could also be `v-bind:class=""`.

```
orders: [
  {
    id: 787352,
    date: "Nov 18, 17:38",
    purchasedBy: "John Miller",
    price: "56.98",
    status: {
      label: "In Progress",
      class: "is-warning"
    }
  },
  {
    id:
    ...
    status: {
      label: "Successful",
      class: "is-success"
    }
  },
  {...}
]
```

As you can see, the first order has a `status.class` of `is-warning`, while the second one has `is-success`. You want all of the status to be`<span>` tags, so give them the `.tag` class. However, the status itself is a variable that is depending on each order. Vue gives you a simple way of toggling CSS classes with the `:class=""` attribute binding.

Note: More on **loops in Vue**.

First Vue template: Login page

In this chapter we take the code from the login.html page and convert that into a Vue component, or page if you will. Here it is a page, but technically each page is just another Vue component. Let's get started.

Open up your `login.html` file and copy over everything inside the `<body></body>` tags into the `<template><template>`part of the `login.vue` file. If you visit your `/login` route, you should now see the same page as your static version. Let's make this page a bit more interactive with Vue.

You might have noticed that the header and side navigation are visible, which they shouldn't be. This is because you use `App.vue` as a base for the admin app. For tutorial purposes, we can make an easy fix for this. You should not do this for production code. What you need to do is wrap your `<nav>` and `<section>` with a `<template>` tag and a check if you are on the login page. To accomplish the latter, you can check a global variable named `this.$route`. This is available when importing `vue-router`. If you want to check a specific route, do so with `this.$route.name`.

```
<template v-if="$route.name !== 'Login'">
  <nav>
    <!--navigation code-->
  </nav>
  <section>
    <!--main content-->
  </section>
</template>
<div v-else><router-view/></div>
```

Note: Inside template code (HTML) we can skip `this` and just write out the variable.

If you test your login page now, it should cover the full size of the page.

Now let's focus on the login page. First off you'll create the data object. You want something to hold your form field information and an error object, so you can toggle error messages on the form. Here's what we came up with:

```
data() {
  return {
    form: {
      email: "",
      password: ""
    },
    error: {
      email: false,
      password: false
    }
  }
},
```

Great, the second thing you need to do is connect these with the form code in the `<template>`. On the input elements, you can add `v-model=""` statements, which binds the input value to the `data` object. So, in this case it's `v-model="form.email"` and `v-model="form.password"`. For the errors, you want to show an error-message and high-

light the input with a red border. Bulma has modifier classes that can be used in many situations. For example, the `.is-danger`class is perfect in this case. You can combine an element with the `.help` class combined with `.is-danger` to show a small help, or an error message in red text.

Start by adding the helper element below `<div class="control">`. This could be something like:

```
<p class="help is-danger" v-if="error.email">Oops! Can't find user.</p>
```

Then add a second one below the `div.control` of the password. To toggle the `.is-danger` class on the inputs, you'll want to make use of Vue's class-bindings. They look like this `:class="{'some-class': someVariable}"`. Both inputs will only have one togglable class. On each `<input>` add, `:class="{'is-danger': error.email}"` and `:class="{'is-danger': error.password}"` respectively.

You are almost done. The only thing that is missing now is submitting the form and checking to see if the values match. For the sake of simplicity, this chapter won't be connecting to a real authentication service. That'll be up to you to implement if you so wish. On the `<button>`, add an event handler: `@click.prevent="tryLogin"`. Down in the `<script>` section, add a new methods object and the `tryLogin()` method.

```
methods: {
  tryLogin(){

  }
}
```

The `tryLogin()` method will do the following:

1. Check if username/password is correct.
2. Show errors if any.
3. Reset possible errors.
4. Send user to Dashboard.

Nothing very fancy is going on here, but you get to see some Bulma classes in action. The finished method looks like this:

```
tryLogin() {
  this.resetErrors();

  if(this.form.email !== 'user@bulma.com'){ return this.error.email = true; }
  if(this.form.password !== 'password'){ return this.error.password = true; }

  this.resetErrors();
  this.$router.push('dashboard');
},
```

```
resetErrors(){
  this.error.email = false;
  this.error.password = false;
}
```

Note: We reset the errors both before and after the `if` checks. This is because you don't want any dangling error messages hanging around after the field has been validated.

This pretty much covers the **Login** component. Hopefully you learned how to show error messages on forms and also toggle a class on elements to highlight errors on input fields.

Creating the "Report a Bug" component

This chapter will recreate the functionality for the **Report a Bug** modal. You can access this modal from the **user menu** in the top-right corner of the topbar. The modal will contain a simple text input and will display a success notification if your imaginary request is successfully completed.

This is what you will be creating:

- Create a `BugReport` component.
- Import the component in the `App.vue` file.
- Add the modal's HTML.
- Add Vue awesomeness.

Creating a component

Let's get going with the first point and create the new component. In the components folder, create a `BugReport.vue` file and start with the following snippet:

```
<template>

</template>

<script>
  export default {
    name: "BugReport"
  }
</script>

<style>

</style>
```

You can go ahead and copy the code for the **Modal card** from the Bulma documentation and insert it between the `<template></template>` tags. Add a nice heading inside

the `.modal-card-title` tag. Inside the `.modal-card-body` you have the `input` and notification.

```
<div class="notification is-success" :class="{'is-hidden': hideNotifica-
tion}">
  <p>
    <span class="icon"><i class="fa fa-bug"></i></span>
    Thanks. Your bug has been reported.
  </p>
  <p>We will do our best to fix it as soon as possible</p>
</div>

<p class="help" :class="{'is-hidden': hideNotification}">The following mes-
sage was sent</p>
<textarea class="textarea" placeholder="Let us know what problems you
faced." :disabled="!hideNotification" v-model="reportMessage"></textarea>
```

There are a few things going on here. The notification makes use of Vue’s class-attribute binding `:class=""`, which we’ve discussed earlier. If the variable **hideNotification** is true then set the class `.is-hidden` to the notification wrapper, and also the little help text above the `<text-area>`. Likewise, `textarea` also uses this variable, but when the *opposite* is true. So when **hideNotification** is false, it’s assumed that the bug-report has been sent and that the success notification is displayed. When it is, the help-text is displayed and the `textarea` is disabled. So the user won’t be able to type any new text.

And finally, the `textarea` has a `v-model` for data-binding. This is so that you can grab that text from the data object and send it off to where it needs to go.

Let’s create the data-objects you’ll need for the `BugReport` component.

```
export default {
  name: "BugReport",
  data() {
    return {
      reportMessage: "",
      hideNotification: true,
    }
  }
}
```

Since this component will be used for other components, it will be the “parent” component’s responsibility to open the modal. As you may know, Bulma modals are shown by toggling the `.is-active` modifier class. You can achieve this be sending down a property from the parent, and if this property is true, you will toggle the `is-active` class. First let’s modify the `<script>` to incorporate the incoming props.

```
export default {
  name: "BugReport",
  props: {
    showModal: {
      type: Boolean,
      default: false
    }
  },
  data() {
    return {
      reportMessage: "",
      hideNotification: true,
    }
  },
}
```

Secondly, use the same class attribute binding as above to toggle the `.is-active` class on the `.modal` wrapper.

```
<div class="modal" :class="{'is-active': showModal}">
  <!-- Modal code -->
</div>
```

Now that it is possible to open and show the modal, you'll want to make sure it can close or be dismissed too.

There are three ways to close the modal:

- Clicking outside the modal (the dark background).
- Clicking the close icon.
- Submitting or cancelling the bug-report.

You do not want to duplicate code, so make a `closeModal()` method, which will be responsible for closing the modal. Now, whichever way you choose to close it, a simple call to the `closeModal()` method will get the job done.

You need to let the parent know that the modal should be closed. Given this, you need to change the `showModal`'s property from `true` to `false`. Communication from child to parent is done through events in Vue. This gives you the following `closeModal()` method, where you can simply `$emit` a close event that the parent handles.

```
closeModal() {
  this.$emit('close');
}
```

The first way to close the modal is implemented in the same fashion. On the `.modal-background` element and the `.delete` button, you can simply add a `@click="closeModal"` handler.

Note: The at sign `@` is short hand for `v-on:`. So the above click event could be `v-on:click="closeModal"`.

For the cancel and submit buttons, create the new functions for sending the bug-report and resetting the `textarea`. You can start with the `resetModal()` method, because it will also be used by the `sendReport()` method.

```
resetModal() {
  this.reportMessage = "";
  this.closeModal();
},
```

First, set the `reportMessage` variable to an empty string and then call the `closeModal()` method from earlier. The second method sends the bug-report as follows.

```
sendReport() {
  /* Do some ajax request to send and save data. */
  this.hideNotification = false

  setTimeout(() => {
    this.hideNotification = true;
    this.resetModal();
  }, 4000);
},
```

What’s happening here is that the status of `hideNotification` changes to false so the notification will show up. To make it a bit more interactive, put in a `setTimeout()` of four (4) seconds, after which you hide the notification again and call the `resetModal()`method.

The final thing to do is add click events on the buttons.

```
<button class="button is-text" @click="resetModal">Cancel</button>
<button class="button is-success" @click="sendReport">Send</button>
```

Our modal is done!

Add the Modal to the App Template

Now that your modal itself is done, you can make it functional from the top bar user menu. Switch over to the `App.vue` file. Next, import the new component and add it to the components object.

```
import BugReport from './components/BugReport.vue';

export default {
```

```
    name: 'app',
    components: { BugReport },
    data: function() {
      return {
        openBugReport: false
      }
    }
  }
```

Then add the component to the bottom of the HMTL template, above the last `</div>`.

```
<report-bug :showModal="openBugReport" v-on:close="openBugReport = false"></
report-bug>
```

Here you can see we are passing along the value of `openBugReport` to the `:showModal` property attribute. You remember how that is the prop that you check for in the `BugReport` component. Your code should also listen for the `close` event that you emitted from the `closeModal()` method earlier. When that happens, the application sets `openBugReport` to `false`, so the modal closes.

Lastly, add a click-handler on the "Report Bug" link. Change this piece of code in the `usermenu`.

```
<a class="navbar-item">
  <span class="icon is-small">
    <i class="fa fa-bug"></i>
  </span> Report bug
</a>
```

So it instead looks like this:

```
<a class="navbar-item" @click="openBugReport = true">
  <span class="icon is-small">
    <i class="fa fa-bug"></i>
  </span> Report bug
</a>
```

Add the **collectjs package** to the project so that you can easily work with arrays and objects.

Books page

We gave you some homework from the `Home.vue` page for this next part, which is for `Books.vue` and the rest of the listings pages. We hope you have managed to create "data-

objects" for the books on this page, and here is a small snippet of how this looks. For the complete code check **the books github repo**.

```
data() {
  return {
    books: [
      {
        name: "TensorFlow For Machine Intelligence",
        price: "$22.99",
        pageCount: 270,
        ISBN: "9781939902351",
        coverImage: "../assets/images/tensorflow.jpg",
        publishDate: 2017,
      },
      {
        name: "Docker in Production",
        price: "$22.99",
        pageCount: 156,
        ISBN: "9781939902184",
        coverImage: "../assets/images/docker.jpg",
        publishDate: 2015,
      },
    ],
    allBooks: []
  }
}
```

The book's page has some simple functionality for filtering and sorting the books on the page. For simplicity sake, you will have two arrays of books in the data-object to start with: **books** and **allBooks**. The latter is just the original array of books you started with when the page loads.

Next, add the **collect.js package** to the project so that you can easily work with arrays and objects. If you have ever worked with the Laravel PHP-framework, you will be very familiar with this package. It is almost an exact JavaScript port of Laravel's Collections.

Sorting books

Sorting the books is really easy, but let's start with importing the `collect.js` package at the top of your `<script>` block.

```
import Collect from "collect.js";
```

There needs to be a way to keep tabs on when a change is made on the select dropdown. With Vue, it is very simple to add event listeners right in your HTML either with the `v-on:event=""` attribute or with shorthand: `@event=""`.

So let's change the `select` element to look like the snippet below:

```
<select @change="sortBooks">
  <option value="publishDate">Publish date</option>
  <option value="price">Price</option>
  <option value="pageCount">Page count</option>
</select>
```

Notice that we also added explicit value attributes to all the options.

The next step is to create the **sortBooks** method and sort the books. Inside the method, use `collect.js` and the `sortBy(key)` method, which simply sorts the collection by the given key.

First, save the currently selected `options` value in a new variable: `let selectValue = String(event.target.value);`

Next, transform the books array into a **Collection**, so that the package can do its magic with the objects. Then create a new collection with the sorted books and finally set that as our books array. Here is the complete **sortBooks** method:

```
sortBooks(event) {
  let selectValue = String(event.target.value);
  let collection = Collect(this.books);
  let sortedBooks = collection.sortBy(selectValue);

  this.books = Object.assign([], sortedBooks.all());
},
```

Filtering books

Now that you have the sorting out of the way, let's see if we can make the filtering/searching just as simple.

And yes, it is even simplier than our sorting above. Let's start with adding event handlers to the `Search` button and to the `<input>` field itself, which will trigger on keyup to make it seem more "active". The `<input>` field will also require a `v-model` attribute for data binding.

```
<p class="control">
  <input class="input" type="text" placeholder="Book name, ISBN…"
    v-model="searchWord" v-on:keyup="searchBooks">
</p>
<p class="control">
  <button class="button" @click="searchBooks">Search</button>
</p>
```

The search button click and keyup will trigger the same method, which does the filtering. One caveat here to keep in mind is that this will change the casing of both the `search-`

`Word` and the `bookname`, so that your `filtersearch` will be somewhat case-insensitve. Besides that, it will run the books array through a Vanilla JS `filter()` method and return the books, which name includes the `searchWord` method.

```
searchBooks() {
  if (!this.searchWord) {
    this.books = Object.assign([], this.allBooks);
  } else {
    this.books = this.books.filter((book) => {
      return book.name.toLowerCase().includes(this.searchWord.toLowerCase());
    });
  }
}
```

Also don't forget to add the **searchWord** to the data object.

```
data() {
...
    coverImage: "../assets/images/gulp.jpg",
      publishDate: 2014,
    },
  ],
  searchWord: "",
...
}
```

Creating and editing a book

The final part of the `books` pages is creating a new book and editing the ones in the list. Again, the focus of this book is Bulma and not VueJS, so this will be a brief and simple explanation on how to do it. Implement a modal on the page. This modal will will open a form so the user can add a new book. This modal could also be used to edit a book as well. This chapter won't go over that. However, if you want to add that functionality yourself, go for it! There is an empty method for this with some notes to get you started.

ADD A NEW BOOK

To start you shoud copy the `ModalCard` code from Bulma and add to the `<template>` part before the last closing `</div>`. Inside the `<div class="modal-card-body">`, paste the `<form></form>` from the `new-book.html` page. Now you should have the base HTML ready to go.

Start by making sure you can open up the modal. To make a Bulma modal visible, it needs the `is-acitve"` modifier class. At this point, the modal only has the `modal` on it as it should, because you don't want it to display all of the time. There are two main ways to

show/hide this modal with Vue. The first is to just include the `is-active` class on the modal by default and show/hide it by toggling the elements `v-show=""` or `v-if=""` attribute.

The `v-if` method might be preferable here because it will remove the markup from the DOM when it is set to false. But let's do it another way by toggling the `is-active` class itself. On the modal wrapper, change the code to the following:

```
<div class="modal" :class="{'is-active': showNewModal}">
```

What is happening here is that we are using Vue's `v-bind:` directive and hooking it into the class attribute. When `showNewModal` is `true`, the `is-active` class is added to the modal `div`. Now set the `showNewModal` variable in your data object with a default value of `false`: `showNewModal: false`. Then add a click event on the new book button. You should now be able to open the modal by clicking the green "New Book" button.

```
<a class="button is-success" @click="showNewModal = true">New</a>
```

So I guess you noticed that there is one tiny little problem--there's no way to close the modal, so let's fix that. There should be a `div` with n class of `.modal-background` on the line after your opening tag for the modal, which is the black background of the modal component. This is a great place to add a second click-event to close out the modal.

You can do this:

```
<div class="modal-background" @click="showNewModal = false"></div>
```

The problem with this is that it will not clear the fields. Instead, you should have a `resetNewBookForm()` method. You'll create this soon, so for now, let's just change the code to:

```
<div class="modal-background" @click="resetNewBookForm"></div>
```

And inside the `methods:` object, create this method to close the modal:

```
resetNewBookForm() {
  this.showNewModal = false;
}
```

Now that you have that in place, let's focus on the input and saving the new book. Again let's use use `v-model` to get the value bound to the data. Let's create a new empty `data` object variable: `book: {}`.

On each `<input>` element on the form, add a `v-model="[input-variable]"`. Where `[input-variable]` corresponds to one of `title`, `price`, `pageCount`, and `ISBN` on the book object. For `publishDate` and `coverImage` you should hard code these on the `saveBook()` method, since we won't be covering uploads in this book.

Each of the inputs should look something like this:

```
<input class="input" type="number" placeholder="e.g. 22.99" value="" re-
quired v-model="book.price">z
```

At the bottom of the form, remove the save and clear buttons, using the ones on the modal.

Here is the finished modal footer:

```
<footer class="modal-card-foot">
  <button class="button is-success" type="button"
    @click="saveBook">Save Book</button>
  <button class="button" type="cancel">Cancel</button>
</footer>
```

The final thing to do is set the static variables as mentioned above, and then use the array `push()` method to add the new book object to the book array(s). Yes it is plural, because we want to add it to both the "original" array of books, `allBooks`, and the one currently in view, `books`.

```
saveBook() {
  this.book.publishDate = "2017";
  this.book.coverImage = "../assets/images/newbook.jpg";

  this.allBooks.push(this.book);
  this.books.push(this.book);

  this.resetNewBookForm();
},
```

And now if you try adding a new book you should see it appear on the page.

REMOVE A BOOK

To remove a book, just remove it from the arrays of book objects.

First, add the click-event to the `.delete` link. You'll pass along the books index in the array: `<a @click="removeBook(index)">Delete</a>`. Then create the `removeBook()` method inside it, simply by splicing the books array from the index.

```
removeBook(index) {
  this.books.splice(index, 1)
},
```

Note: In a fully fledged application, you should extract the modal into it's own component to be re-used across your application. Switching the content inside a modal can be done, for example, with Vue's `<slot>`.

Summary

That wraps up this chapter on integrating Bulma with VueJS. As mentioned in the beginning, there are some snippets of the code up on **GitHub** where you can start to implement the editing form for a book. You can either create a new edit-book page or use a modal like we did here to add a new book.

The next chapter covers using Bulma with React.

12 Using Bulma with React

In this chapter, you will be integrating Bulma with React. React is a popular JavaScript framework created by Facebook to create user interfaces.

```
React Documentation: https://reactjs.org
```

Before you dive too deep into this chapter, we should go over some of the prerequisites and expectations. You should have some basic knowledge of JavaScript (ES6), React (or React Native), Create React App (React CLI), React Router, and NPM or Yarn (Facebook's package manager). For this chapter however, we will be using NPM.

What you will be making

In this chapter you will be making a collection browser for Bleeding Edge Press. For this app, users will be able to login with an email and password, view a collection of books and view book details.

Note: This chapter will also use React best practices and best practice naming conventions. It's worth nothing that this chapter will *not* be using state management with Redux or tips of server side rendering. Instead it will focus more on the user interface.

Overall, it is a pretty simple application. By the end of this chapter, you should know how to properly integrate Bulma with React and leverage the Bulma library to create your application's user interface.

Installing "Create React App"

Much like Angular and VueJS, React also has it's own CLI; it's called Create-React-App. Before we start creating our interface with Bulma, you'll need to jump start the React application by running a few commands.

```
npm install -g create-react-app
create-react-app <project-name>
```

```
cd <project-name>
npm start
```

A local server will start and your React app will initialize.

Quick overview of Create-React-App

Create-React-App already does all of the hard work of setting up a development environment for you. From here, you just need to create the components and stylesheets (if any).

For this chapter, we're going to keep all components and their children in their own directory.

```
src/
- components/
  - Login/
    - Login.jsx
    - LoginForm.jsx
    - styles/ (if any)
      - Login.css
```

`Login.jsx` will act as the container with `LoginForm.jsx` nested inside it. Setting up the components this way will let you move or add your login form anywhere in the application.

The app structure

You are going to be renaming some files, creating directories for your assets, and creating directories for your components. At a very high level, your directory structure inside the `src` folder should resemble this...

```
src/
  - assets/
  - actions/
  - components/
    - ComponentName/
      - ComponentName.jsx
      - ComponentNameChild.jsx
      - ComponentNameOtherChild.jsx
      - styles/
        - ComponentName.css
    - App.css
    - App.js
    - App.test.js
  - index.js
```

```
- index.css
- registerServiceWorker.js
```

Installing Bulma

There are a few ways that you can initialize Bulma inside the React app. You can certainly add it to your `index.html` file inside the `_public/` directory, or...you can add it via NPM and import it with ES6.

Note: You will want to add Bulma globally to refer to it once and use it throughout the entire application.

Option 1: Adding Bulma via a CDN

After Create-React-App is done installing, start the application with **npm start** and open the files in a text editor. In your project structure, you will see a `public/` directory. Navigate to the `public/` directory and open the `index.html` file.

You can remove the pre-rendered comments if you'd like, but these aren't too important.

Inside the <head>, add Bulma via a CDN just like any other stylesheet in a website.

```
<link href="https://cdnjs.cloudflare.com/ajax/libs/bulma/0.6.2/css/
bulma.min.css" rel="stylesheet">
```

Option 2: Adding Bulma via NPM

This is the recommended way of adding Bulma, since it's considered best practice to import React dependencies with JavaScript.

After Create-React-App is done installing, start the application with **npm start** and open the files in a text editor.

To install Bulma via NPM run...

```
npm install bulma --save
```

Let's open up the `index.js` file inside our main `src/` directory and add the following with the rest of the import statements...

```
import './../node_modules/bulma/css/bulma.css';
```

That's it. You can start using Bulma in your JSX!

Routing with React Router 4

This example application is using React Router 4, which allows you to visit different rendered components based on the URL. This chapter will briefly go over the basics of React Router 4. However, it's strongly recommended to check out their documentation.

React Router 4 Documentation: **https://reacttraining.com/react-router/**.

First, you'll want to install React Router 4 with the following command:

```
npm install react-router-dom --save
```

Next, import two specific components of `react-router-dom` and those are `BrowserRouter` and `Route`. You can do that with an ES6 import statement in your `App.js` file.

```
import { BrowserRouter, Route } from 'react-router-dom';
```

Next, import your components that you will create. You can actually import these later once you create them. If you import them before, you'll get an error. Just be sure to reference this section later when you're ready to tie routes to components.

BrowserRouter

`<BrowserRouter>` is a wrapper for each `<Route>`. Think of `BrowserRouter` as a component itself that get's it's "child" component injected into when a certain condition is met like, let's say...a URL address?

Like all other components, `<BrowserRouter>` needs a single root element. You will get an error if you try to place many routes in there. So, you'll need to follow it directly with a `<div>`.

At this point, your JSX should look something like this:

```
<BrowserRouter>
  <div>
    {/* Routes will go here */}
  </div>
</BrowserRouter>
```

Route

Inside your single `<div>`, you should add a `<Route>`. Remember, `<Route>` is a component that was imported with React Router 4. The basic structure of a route is:

```
<Route exact path="/" component={Login} />
```

At some point in this chapter, we will be creating a dynamic route. Dynamic routes have variables that you can add to the route in order to assign a unique route to a component with unique data.

Variables in dynamic routes begin with a colon `:` followed by the variable name like, `id`.

Final App.js With Routes

```
import React, { Component } from 'react';
import { BrowserRouter, Route } from 'react-router-dom';
import './App.css';

// Import Components for Routes
import Login from './Login/Login';
import Collection from './Collection/Collection';
import CollectionSingleBookDetail from './Collection/CollectionSingleBookDe-
tail';

class App extends Component {
  render() {
    return (
      <BrowserRouter>
        <div>
          <Route exact path="/" component={Login} />
          <Route exact path="/collection" component={Collection} />
          <Route name="collectionDetail" path="/collection/:id" compo-
nent={CollectionSingleBookDetail} />
        </div>
      </BrowserRouter>
    );
  }
}

export default App;
```

Creating the Login component

Let's create a **folder** and name it "Login." As stated above, this folder will contain all of our component's code. Inside this folder, let's create a **JSX file** and name it "Login.jsx".

This `Login.jsx` will act as a container and will do nothing but "contain" the child components. On this level, you can control the overall layout of the component. You want to keep UI layout separate from the child components. If it's confusing now, don't worry, it'll make more sense soon.

Login.jsx

Remember, Bulma was added globally into the `index.js` file earlier. So you don't need to add it again, so let's create the first React component with Bulma.

CREATING THE LOGIN FORM CONTAINER

First, create the user interface that *contains* the form. Remember, you should keep the actual form *separate* from the `Login` component. That way you can reuse the form itself anywhere in the web application if you choose to do so.

For every new component we want to import a few things into it using ES6 and then render our component. All of our JSX will be written inside the component that extends `Component`.

```
import React, { Component } from 'react';

class Login extends Component {
  render() {
    return (
      {/* JSX Goes Here */}
    );
  }
}

export default Login;
```

Bulma has some nice utility classes out-of-the-box that you can leverage to create that full height "green" background. In order to achieve that, you'll want to create a single element and assign it a few classes: a base class and two modifiers.

Tip: Modifier classes in Bulma begin with `is-` or `has-`.

Those classes are:

- `hero`: Defines a large area for hero images or important information.
- `is-primary`: Adds the `primary` background color. In Bulma, the primary color is the green color we want.
- `is-fullheight`: Applies a minimum height of 100% of the viewport's height.

```
<section className="hero is-primary is-fullheight">

</section>
```

Your browser window should be completely green. However, if you add some arbitrary content, you'll notice that it's not vertically aligned. Luckily, there's a Bulma class that does this, specifically used in tandem with the `hero` class, and that is `hero-body`.

```
<section className="hero is-primary is-fullheight">
  <div className="hero-body">
    <p>I am generic text.</p>
  </div>
</section>
```

If you add some generic text now, you'll notice that your text is now vertically centered! You still need to add a few more lines of JSX with Bulma classes to achieve the container's desired user interface.

```
<section className="hero is-primary is-fullheight">
  <div className="hero-body">
    <div className="container">
      <div className="columns is-centered">
        <div className="column is-5-tablet is-4-desktop is-3-widescreen">
          {/* Our form code goes here */}
          <p>I am generic text.</p>
        </div>
      </div>
    </div>
  </div>
</section>
```

- `container`: Contains the child elements in a pre-defined width.
- `columns`: The "row" that contains our individual columns.
- `is-5-tablet`: Column is 5/12 columns wide on tablet devices.
- `is-4 desktop`: Column is 4/12 columns wide of desktop devices.
- `is-3-widescreen`: Column is 3/12 columns wide on larger, widescreen devices.

FINAL LOGIN.JSX

```
import React, { Component } from 'react';

class Login extends Component {
  render() {
    return (
      <section className="hero is-primary is-fullheight">
        <div className="hero-body">
          <div className="container">
            <div className="columns is-centered">
              <div className="column is-5-tablet is-4-desktop is-3-
widescreen">
                <p>I am generic text.</p>
              </div>
            </div>
          </div>
```

```
          </div>
        </section>
      );
    }
  }

  export default Login;
```

Creating the Login form

Now that the container is complete, let's create the login form itself. This `LoginForm.jsx` component will be imported into `Login.jsx` as a child component.

`LoginForm.jsx`

```
import React, { Component } from 'react';
import Logo from './../../assets/logo-bis.png'; {/* Logo Image */}

class LoginForm extends Component {
  render() {
    return (
      {/* JSX Goes Here */}
    );
  }
}

export default LoginForm;
```

Most of this JSX is standard form inputs and checkboxes. Let's add this JSX into your `return` statement in the `LoginForm` component.

Every form needs a few things, but most importantly, the `<form>` element. This form element will be our single root element in this example. We're going to give it a Bulma class of box. What box does is adds a white background with a slight drop shadow to our form.

```
<form className="box">

</div>
```

Next, add the logo. If you haven't already, import the logo with an ES6 import statement. The image will be wrapped with some Bulma classes to a `<div>` so it can be centered at the top of the form. The Bulma class, `has-text-centered` does just that.

```
<div className="field has-text-centered">
  <img src={Logo} width="167"/>
</div>
```

From here, it's just a matter of creating the rest of the form inputs for the email and password fields as well as the submit button. As you can probably guess, we are going to be leveraging Bulma for our input fields.

```
<div className="field">
  <label className="label">Email</label>
  <div className="control has-icons-left">
    <input className="input" type="email" placeholder="e.g. dave@parsecdigi-
tal.io" required/>
    <span className="icon is-small is-left">
      <i className="fa fa-envelope"></i>
    </span>
  </div>
</div>
```

You'll notice a few extra classes like `label`, `has-icons-left`, `is-small`, and `is-left`. These are used to get the styling of our form consistent. More importantly though, `has-icons-left` tells the form input that it is supposed to have icons to the left of the input. So, with that class, Bulma adds some padding to leave room for an icon.

Note: This form is using Font Awesome, which are text SVG icons. As the name suggests, it's pretty awesome. You should definitely check out their documentation.

Font Awesome Documentation: **http://fontawesome.io/**

```
<form className="box">
  <div className="field has-text-centered">
    <img src={Logo} width="167"/>
  </div>
  <div className="field">
    <label className="label">Email</label>
    <div className="control has-icons-left">
      <input className="input" type="email" placeholder="e.g. dave@parsecdi-
gital.io" required/>
      <span className="icon is-small is-left">
        <i className="fa fa-envelope"></i>
      </span>
    </div>
  </div>
  <div className="field">
    <label className="label">Password</label>
    <div className="control has-icons-left">
      <input className="input" type="password" placeholder="********" re-
quired/>
      <span className="icon is-small is-left">
        <i className="fa fa-lock"></i>
      </span>
    </div>
  </div>
```

```
    <div className="field">
      <label className="checkbox">
        <input type="checkbox" required/>
        Remember me
      </label>
    </div>
    <div className="field">
      <button className="button is-success">
        Login
      </button>
    </div>
  </form>
```

- box: Adds a white box to contain child elements.
- `field`: Contains `<form>` elements for consistent spacing.
- `control`: A form input container.
- `has-icons-left`: Adds padding to left of input field to allow room for an icon.
- `input`: Styling for form inputs.
- `is-small`: Modifier that decreases the size of the element.
- `is-left`: Aligns the icon to the left.
- checkbox: Styling for form checkboxes.

Note: It's worth noting that we are not going to be adding validation or a form handler to this form. This section is to illustrate how easy it is to create a web form with Bulma.

Optional: Feel free to add validation and a form handler. Write a function that redirects the user to the `/collections` route when submitted correctly.

FINAL LOGINFORM.JSX COMPONENT

```
import React, { Component } from 'react';

class LoginForm extends Component {
  render() {
    return (
      <form className="box">
        <div className="field has-text-centered">
          <img src={Logo} width="167"/>
        </div>
        <div className="field">
          <label className="label">Email</label>
          <div className="control has-icons-left">
            <input className="input" type="email" placeholder="e.g. dave@par-
secdigital.io" required/>
            <span className="icon is-small is-left">
              <i className="fa fa-envelope"></i>
```

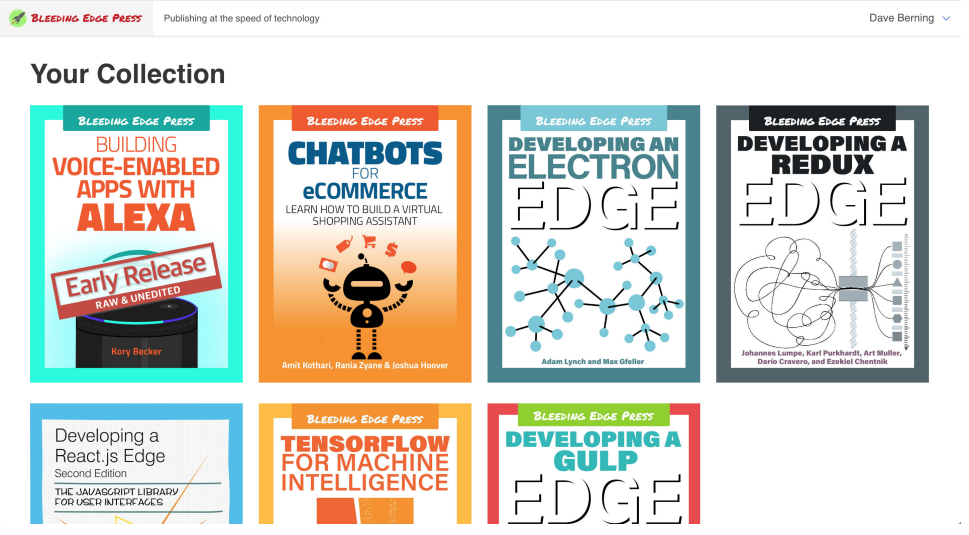
```
            </span>
          </div>
        </div>
        <div className="field">
          <label className="label">Password</label>
          <div className="control has-icons-left">
            <input className="input" type="password" placeholder="********"
required/>
            <span className="icon is-small is-left">
              <i className="fa fa-lock"></i>
            </span>
          </div>
        </div>
        <div className="field">
          <label className="checkbox">
            <input type="checkbox" required/>
            Remember me
          </label>
        </div>
        <div className="field">
          <button className="button is-success">
            Login
          </button>
        </div>
      </form>
    );
  }
}

export default LoginForm;
```

After your form is complete, import it into your `Login.jsx` component with `import LoginForm from './LoginForm.jsx';` and replace your generic text with `<LoginForm />`.

You can further enhance this form on your own with validation, and by routing the Collections component when valid.

Creating the collection

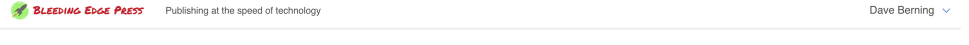
Once you “login”, you’ll be “redirected” to a collections view. This collection is the meat-and-potatoes of this chapter’s example. The collection will display various Bleeding Edge book covers. Users can click on a cover and get routed to a “details” component where they cab “buy” or “share” the book.



The Header

Every web application needs a header. As you've probably guessed, Bulma has classes that we can use that make this very easy.

You're final header should look something to this when done:



Create your `Header.jsx` component and place the JSX file in the `src/components/Header/` directory. This `Header` component will be the header "container".

Header.jsx

The base element for this component will be `<header>`. Inside that `<header>` will be a `<nav>` with some Bulma classes.

```
<header>
  <nav>

  </nav>
</header>
```

This header is great so far, but we want some separation between it and the rest of the webpage. So, add the `has-shadow` class to the `<nav>` to add a slightly drop shadow. You should also add the `navbar` class to add the default nav bar styles in Bulma.

Your header's JSX should resemble this. Don't worry about the `HeaderBrand` and `HeaderUserControls`. You'll make those soon.

Next, it's time to add the JSX for the nav bar itself. As you can see, `navbar-menu` is needed for the navigation bar especially for the desktop. The reason is that if you want to show a navigation bar on *desktop*, but not *mobile*, use the `navbar-menu` class.

`Navbar-start` is used for the *left* section of the navigation bar. `Navbar-item` is used to define each *single* item in the navigation bar.

```
<div className="navbar-menu">
  <div className="navbar-start">
    <div className="navbar-item">
      <small>Publishing at the speed of technology</small>
    </div>
  </div>
</div>
```

FINAL HEADER.JSX

```
import React, { Component } from 'react';
import HeaderBrand from './HeaderBrand';
import HeaderUserControls from './HeaderUserControls';

class Header extends Component {
  render() {
    return (
      <header>
        <nav className="navbar has-shadow">
          <HeaderBrand />

          <div className="navbar-menu">
            <div className="navbar-start">
              <div className="navbar-item">
                <small>Publishing at the speed of technology</small>
              </div>
            </div>
            <HeaderUserControls />
          </div>
        </nav>
      </header>
    );
  }
}

export default Header;
```

- `navbar`: Full width, responsive vertical navigation bar with a structure; the main container.
- `has-shadow`: Modifier that adds a box-shadow to the element.
- `navbar-start`: The left part of the menu, which appears next to the navbar brand on desktop.
- `navbar-item`: Each single item of the navbar, which can either be an `a` or a `div`.

HeaderBrand.jsx

`HeaderBrand` is a child component of `Header` and is used for our branding, including the logo!

Create a new file and name it `HeaderBrand.jsx`, placing it inside your `src/components/Header` directory. Once you import React, make sure you import the logo as a component dependency.

Our base element is going to be a `<div>` with some Bulma clases attached to it.

You should wrap your logo image with `navbar-item` and `navbar-brand`, as shown below. The class, `navbar-brand` is used because it is *always* visible across all devices. This class is typically used for branding, like for logos or mottos.

```
<div className="navbar-brand">
  <a className="navbar-item">
    <img src={Logo} />
  </a>
</div>
```

Next in this component, create the mobile navigation icon for mobile devices. Bulma makes this very easy. Create three `<span>` tags and wrap those in `navbar-burger` and `burger`.

```
<div className="navbar-burger burger">
  <span></span>
  <span></span>
  <span></span>
</div>
```

Hamburger icons have never been so easy!

FINAL HEADERBRAND.JSX

```
import React, { Component } from 'react';
import Logo from './../../assets/logo.png';

class HeaderBrand extends Component {
```

```
  render() {
    return (
      <div className="navbar-brand">
        <a className="navbar-item">
          <img src={Logo} />
        </a>
        <div className="navbar-burger burger">
          <span></span>
          <span></span>
          <span></span>
        </div>
      </div>
    );
  }
}

export default HeaderBrand;
```

- `navbar-brand`: Always visible and usually contains the logo and optionally some links or icons.
- `navbar-burger`: The hamburger icon, which toggles the `navbar` menu on touch devices.
- `burger`: We wish. Jokes aside, this is a container that contains the three `<span>` tags that'll render a hamburger, mobile navigation icon.

HeaderUserControls.jsx

`HeaderUserControls.jsx` is our final component for the header. This is just a simple dropdown with additional links for things like "Profile" and "Sign Out". Create a new file and name it `HeaderUserControls.jsx` inside of our `src/components/Header` directory. The base element of this component is going to be a `<div>`.

```
<div className="navbar-end">

</div>
```

`Navbar-end` is used because this will be at the end or the *right* of our navigation bar. Add a nest `<div>` inside of it and assign the Bulma classes `has-dropdown` and `is-hoverable`. You can probably already guess what these modifiers do. It makes it easy to create a dropdown that displays on hover.

```
<div className="navbar-end">
  <div className="navbar-item has-dropdown is-hoverable">
    <div className="navbar-link">
      Dave Berning
```

```
      </div>
    </div>
  </div>
```

As you can see, this a great but there is no dropdown. So you'll need to create that next. Your dropdown menu should *always* be wrapped in the `navbar-dropdown` class. With each dropdown item, make sure you wrap them with the `navbar-item` class.

```
<div className="navbar-dropdown">
  <a className="navbar-item">

<div>
      <span className="icon is-small">
        <i className="fa fa-user-circle-o"></i>
      </span>
      Profile
    </div>
  </a>
  <a className="navbar-item">
    <div>
      <span className="icon is-small">
        <i className="fa fa-bug"></i>
      </span>
      Report bug
    </div>
  </a>
  <a className="navbar-item">
    <div>
      <span className="icon is-small">
        <i className="fa fa-sign-out"></i>
      </span>
      Sign Out
    </div>
  </a>
</div>
```

The JSX for the dropdown should be under the `navbar-link` with user's name. In this case, "Dave Berning."

- `navbar-end`: The right part of the menu, which appears at the end of the navbar.
- `is-hoverable`: The dropdown will show up when hovering the parent `navbar-item`.
- `navbar-link`: A link as the sibling of a dropdown, with an arrow.

FINAL HEADERUSERCONTROLS.JSX

Your final `HeaderUserControls` component should look similar to this:

```
import React, { Component } from 'react';

class HeaderUserControls extends Component {
  render() {
    return (
      <div className="navbar-end">
        <div className="navbar-item has-dropdown is-hoverable">
          <div className="navbar-link">
            Dave Berning
          </div>
          <div className="navbar-dropdown">
            <a className="navbar-item">

<div>
                <span className="icon is-small">
                  <i className="fa fa-user-circle-o"></i>
                </span>
                Profile
              </div>
            </a>
            <a className="navbar-item">
              <div>
                <span className="icon is-small">
                  <i className="fa fa-bug"></i>
                </span>
                Report bug
              </div>
            </a>
            <a className="navbar-item">
              <div>
                <span className="icon is-small">
                  <i className="fa fa-sign-out"></i>
                </span>
                Sign Out
              </div>
            </a>
          </div>
        </div>
      </div>
    );
  }
}

export default HeaderUserControls;
```

Putting the header together

Now that you have your header child components done, it's time to import them into `Header.jsx`. Your final header should resemble something like this:

```
import React, { Component } from 'react';
import HeaderBrand from './HeaderBrand';
import HeaderUserControls from './HeaderUserControls';

class Header extends Component {
  render() {
    return (
      <header>
        <nav className="navbar has-shadow">
          <HeaderBrand />

          <div className="navbar-menu">
            <div className="navbar-start">
              <div className="navbar-item">
                <small>Publishing at the speed of technology</small>
              </div>
            </div>
            <HeaderUserControls />
          </div>
        </nav>
      </header>
    );
  }
}

export default Header;
```

Footer.jsx

The Footer is a much simpler component than the header. However, you can certainly try out your new Bulma skills and add additional columns, images, text, and a footer navigation bar.

Create a new JSX file and name it `Footer.jsx`, placing it into the `src/Footer/` directory.

Your JSX for this is very simple:

```
<footer className="footer">
  <p className="has-text-centered">Copyright &copy; 2018. All Rights Reserved</p>
</footer>
```

- `footer`: Used for footers. You can have any element, list, or image in this element.
- `has-text-centered`: Center aligns text.

You have your footer already constructed. Later, you'll import the header and footer into the collections and collections detail components.

Your final footer component should resemble this:

```
import React, { Component } from 'react';

class Footer extends Component {
  render() {
    return (
      <footer className="footer">
        <p className="has-text-centered">Copyright &copy; 2018. All Rights
Reserved</p>
      </footer>
    );
  }
}

export default Footer;
```

The book collection body

This body will control the layout of the collection as well as iterate through data and render a single component that you'll pass data into. For this section, the data is coming from a JSON file with generic data called, `books.json` in the `src/data` directory.

The data object looks something like this:

```
{
  "id": 5,
  "name": "Developing a React.js Edge",
  "cover": "react-edge.jpg",
  "author": "Richard Feldman, Frankie Bagnardi, & Simon Hojberg",
  "details": "Lorem ipsum dolor sit amet..."
}
```

Create a JSX file called `Collection.jsx` and place it into the `src/components/Collection/` directory. This component will act as our container and contain all child components. The base element in this component is going to be a `<div>`. Nested inside of that base `<div>` is going to be another with the class of `container`. This `container` class is used to "contain" our content in a fixed width and be centered.

Collection.jsx

```
<div>
  <div className="container">

  </div>
</div>
```

Next, add some JSX to fill in the component. The end goal of this component is to show all of the book covers with one single component. To achieve this, we want to loop through data, pass props down, and write corresponding JSX with Bulma to achieve this. Create a `<div>` with the class of `columns`. Following that `columns` `<div>` there should be another `<div>` with the class of `column`. This of course is the base of Bulma, which was discussed in other chapters.

You want to iterate through that data and the container `CollectionSingleBook` component with a column that is 3/12 columns wide. When referencing the `CollectionSingleBook` component, make sure you pass down you data via props.

```
<h1 className="title is-2">Your Collection</h1>
{/* Iterates through data (books) */}
<div className="columns is-multiline">
  {this.state.books.map((book) => (
    <div className="column is-3">
      <CollectionSingleBook key={book.id} book={book} /> { /* Creating this
soon! */ }
    </div>
  ))}
</div>
```

FINAL COLLECTION.JSX

```
import React, { Component } from 'react';
import Header from './../Header/Header'
import Footer from './../Footer/Footer';
import CollectionSingleBook from './CollectionSingleBook';
import BookData from './../../data/books.json';
import styles from './styles/Collection.css';

class Collection extends Component {
  constructor() {
    super();
    this.state = {
      books: BookData
    };
  }

  render() {
```

```
    return (
      <div>
        <Header />
        <div className="container has-gutter-top-bottom">
          <h1 className="title is-2">Your Collection</h1>
          {/* Iterates through data (books) */}
          <div className="columns is-multiline">
            {this.state.books.map((book) => (
              <div className="column is-3">
                <CollectionSingleBook key={book.id} book={book} />
              </div>
            ))}
          </div>
        </div>
        <Footer />
      </div>
    );
  }
}

export default Collection;
```

- `title`: Defines a title (much like an `<h1>`)
- `is-2`: Based on a 12 column layout. Element is 2/12 columns wide.
- `is-multiline`: Defines the `columns` row to wrap `column` items. Wihout this, the columns will repeat past it's container without wrapping.
- `is-3`: Based on a 12 column layout. Element is 3/12 columns wide.

CollectionSingleBook.jsx

This is a smaller component. `CollectionSingleBook.jsx` is simply going to be our book cover with a link to the detail component. This component really illustrates why you should break up components into small, digestible bits.

To elaborate, the `CollectionSingleBook` component is restricting the size of the cover to a third of the browser window, or in this case, the container. With no size restriction on the single book itself, you can add it anywhere and control the size using other parent components.

Note: `Link` is part of React Router 4. Import it with `import { Link, withRouter} from 'react-router-dom';`

```
<div>
  <Link to={{pathname: `/collection/${this.props.book.id}`, state: { singleBook: this.props.book }}}>
    <img src={require("./../../assets/" + this.props.book.cover)}/>
```

```
    </Link>
  </div>
```

In this component, you are simply constructing the dynamic link and passing a single "book" object down via props to the next component the `CollectionSingleBookDetail.jsx`.

FINAL COLLECTIONSINGLEBOOK.JSX

```
import React, { Component } from 'react';
import { Link, withRouter} from 'react-router-dom';

class CollectionSingleBook extends Component {
  render() {
    return (
      <div>
        <Link to={{pathname: `/collection/${this.props.book.id}`, state:
{ singleBook: this.props.book }}}><img src={require("./../../assets/" +
this.props.book.cover)}/></Link>
      </div>
    );
  }
}

export default CollectionSingleBook;
```

CollectionSingleBookDetail.jsx

This is a dynamic component. Meaning, the route is always different but uses the same component. The route is what defines which data gets passed down to this. You access this component by clicking on the `CollectionSingleBook.jsx`. We are using the book's id to determine which book info get's loaded into this component.

This component's layout is pretty simple; it's two columns. The left column is nothing but the book cover while the right is nothing but information about the book as well as a nested `columns` row for the "share" and "buy" buttons. Make sure your add the `container` class to the wrapper `<div>` so it restricts the content to a fixed width and centers it.

When you see `singleBook`, it is referencing the data directly via props from `Collection.jsx`. The left column needs to have the `is-one-third` modifier class. You *need* to restrict the width of that column to a certain size otherwise the cover image will be way too large. After that left column, the other sibling columns automatically adjust their size.

```
<div className="container">
  <div className="columns">
    <div className="column">
      <h1 className="title is-2">{singleBook.name}</h1>
```

```
      <p>By: {singleBook.author}</p>
    </div>
  </div>
  <div className="columns">
    <div className="column is-one-third">
      <img src={require("./../../assets/" + singleBook.cover)}/>
    </div>
    <div className="column">
      <p>{singleBook.details}</p>

      <div className="columns">
        <div className="column">
          <button className="button is-primary is-large is-fullwidth">Buy
Book</button>
        </div>
        <div className="column">
          <button className="button is-secondary is-large is-
fullwidth">Share Book</button>
        </div>
      </div>
    </div>
  </div>
</div>
```

Your final component should resemble something similar to this:

```
import React, { Component } from 'react';
import Header from './../Header/Header';
import Footer from './../Footer/Footer';

class CollectionSingleBookDetail extends Component {
  render() {
    const singleBook = this.props.location.state.singleBook; { /* just mak-
ing our JSX easier to read. This is optional. */}

    return (
      <div>
        <div className="container">
          <div className="columns">
            <div className="column">
              <h1 className="title is-2">{singleBook.name}</h1>
              <p>By: {singleBook.author}</p>
            </div>
          </div>
          <div className="columns">
            <div className="column is-one-third">
              <img src={require("./../../assets/" + singleBook.cover)}/>
            </div>
            <div className="column">
```

```
                <p>{singleBook.details}</p>

                <div className="columns">
                  <div className="column">
                    <button className="button is-primary is-large is-
fullwidth">Buy Book</button>
                  </div>
                  <div className="column">
                    <button className="button is-secondary is-large is-
fullwidth">Share Book</button>
                  </div>
                </div>
              </div>
            </div>
          </div>
        );
    }
}

export default CollectionSingleBookDetail;
```

- `title`: Adds heading styles to text.
- `is-2`: Different size of the `.title`. Comparable to a <h2>.
- `is-one-third`: Defines the column to be one-third of the container. Other columns fill in the rest of the space.
- `is-secondary`: Uses the secondary color for the <button>.
- `is-large`: Increases the size of the button to a larger size.
- `is-fullwidth`: Makes the <button> 100% width.

Tying the Collections Component Together

Now that you have all of the collection components completed, you should start importing the headers and footers into your `Collection.jsx` component.

`import Header from './../Header/Header'; import Footer from './../Footer/Footer';`

In your `Collections.jsx` and `CollectionSingleBookDetail.jsx` components, add <Header /> and <Footer /> above and below the `.container` respectively.

Your final code should resemble the following:

Collections.jsx (Container)

```
import React, { Component } from 'react';
import Header from './../Header/Header'
import Footer from './../Footer/Footer';
```

```
import CollectionSingleBook from './CollectionSingleBook';
import BookData from './../../data/books.json';

class Collection extends Component {
  constructor() {
    super();
    this.state = {
      books: BookData
    };
  }

  render() {
    return (
      <div>
        <Header />
        <div className="container has-gutter-top-bottom">
          <h1 className="title is-2">Your Collection</h1>
          {/* Iterates through data (books) */}
          <div className="columns is-multiline">
            {this.state.books.map((book) => (
              <div className="column is-3">
                <CollectionSingleBook key={book.id} book={book} />
              </div>
            ))}
          </div>
        </div>
        <Footer />
      </div>
    );
  }
}

export default Collection;
```

Running the application

If you haven't been building the example throughout this chapter, you should run the following command to build your project locally:

```
npm start
```

Assuming that it builds correctly, you should see the login screen! This form doesn't have any functionality, but to see the Collections component, you'll need to navigate to it via the URL bar with `/collection`.

You should see a grid of Bleeding Edge Book covers! From here, you can click on each book cover and it'll take you to `/collection/<id>`. Each one of these detail screens is a single component that gets data passed into it.

Summary

Bulma is a powerful CSS framework that you can use in your next React project to quickly prototype and or create it's user interface. Since Bulma is built on Flexbox, some of these concepts you can take with you if you decide to build a native mobile application with React Native!

You should now have a strong understanding on how to integrate Bulma with React and why you might decide to use a specific Bulma CSS class in a specific case.

Follow along in the next and final chapter to learn how to customize Bulma.

13 Customizing Bulma

Bulma comes with default styles that are carefully chosen to satisfy most users, and ensure that any interface built with Bulma looks great.

But even if the layout of the page is naturally balanced and the components are clear enough to be used straight out of the box, you probably don't want your website to end up looking like every other Bulma instance. First, because you probably already have defined colors and typography rules, which is especially true when you are using Bulma in a business context, where branding guidelines have already been defined and need to be strictly followed. Secondly, because no matter what the purpose of the website you're building with Bulma, you'll still want to add your own personal touch. And one design can't satisfy everyone!

Luckily, Bulma is a CSS framework that is very easy to customize, and it can be done in several ways:

1. Overriding Bulma's initial and derived variables
2. Overriding Bulma's component variables
3. Adding your own variables
4. Overriding Bulma's styles
5. Adding your own styles

To define your own customized interface, you need to follow at least one of these steps.

Have a look at the **Bulma Expo**, and you'll see that Bulma is a tool that can be the solution to *any* type of design.

First, you need to set up Sass on your computer.

Setting up node-sass

Bulma is built with **Sass**, a CSS preprocessor. Although it's originally written in Ruby (and available as a Ruby gem), it's recommended to use the faster C/C++ compiler **LibSass**.

Most developers actually use **node-sass**, which provides binding for Node.js to LibSass. This is the library we're going to use here.

You need to have **NodeJS** installed on your computer.

Creating package.json

With your terminal, go to the folder where you've saved your HTML files (alongside `books.html`, `customers.html`, etc.), and type the following:

```
npm init
```

Follow the instructions. This will create a `package.json` file.
Then type the following:

```
npm i bulma node-sass --save-dev
```

This will add the dev dependencies to your `package.json`:

```
"devDependencies": {
  "bulma": "^0.6.2",
  "node-sass": "^4.7.2"
}
```

Right now, the list of scripts only has one called `test`, which simply echos an error message and exits.
Replace that script with the following list:

```
"scripts": {
  "build": "node-sass --output-style expanded --include-path=node_modules/
bulma sass/custom.scss css/custom.css",
  "start": "npm run build -- --watch"
},
```

The most important script here is the `build` one: it takes the file `sass/custom.scss` as an **input**, and creates the `css/custom.css` as the **output**.
The `start` script simply turns the `build` into a watch script.

Creating a sass/custom.scss file

So far, you've been importing the Bulma CSS via the CDN:

```
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/bulma/
0.6.1/css/bulma.min.css">
```

Since you want your own custom version, in all of your HTML files **replace** that `<link>` tag with this new one:

```
<link rel="stylesheet" href="css/custom.css">
```

The `/css` folder and the `custom.css` file don't exist yet!

In the same directory as your `package.json` file, create a `/css` and a `/sass` folder. In the latter, add a `custom.scss` file.

While Bulma itself uses `.sass` files, most developers prefer the syntax `.scss` files because it's easier to understand, hence why we're using it here.

To see if your setup is working, write the following in `custom.scss`:

```
html {
  background: red;
}
```

Then open up your terminal and run `npm run build`. You should see the following output:

```
Rendering Complete, saving .css file...
Wrote CSS to /path/to/html/css/custom.css
```

Open up your page and you should see the following:

By using your own `custom.css` file:

- You removed the Bulma styles
- You added your own styles

You can now remove this CSS rule so `custom.scss` is **empty**.

Importing Bulma

You have installed Bulma locally on your machine, but you are not using it yet.

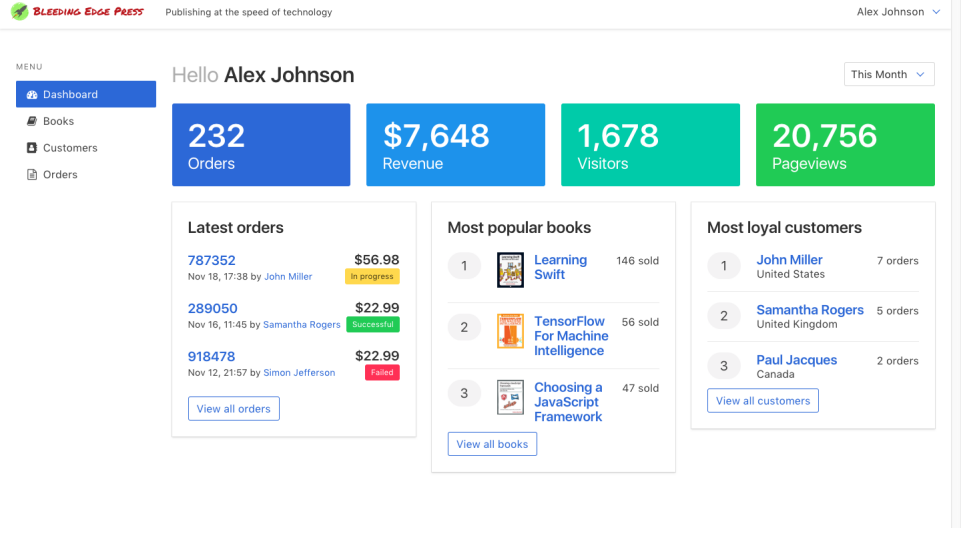
Because you are going to update the `.scss` file quite frequently from now on, run `npm start` instead: this will watch the file for changes.

In your empty `custom.scss` file, add:

```
@import "node_modules/bulma/bulma";
```

Save the file. Since a change in the file has occurred, you will see this output in the terminal:

```
=> changed: /path/to/html/sass/custom.scss
Rendering Complete, saving .css file...
Wrote CSS to /path/to/html/css/custom.css
```



Everything is back to normal. Instead of importing the generated `.css` file from the CDN, you are importing the Sass version of Bulma into your `custom.scss` file, which then generates the `custom.css`.

Since you haven't made any changes yet, you can't see any differences. The first step to create your custom design is to import new font families.

Importing the Google fonts

The new design uses two Google fonts: Karla and Rubik. While it's possible to import them via a `<link>` tag, it's easier to import them in a single location from your CSS file.

You import the fonts *before* importing Bulma. So at the top of `custom.scss` file, add:

```
@import url('https://fonts.googleapis.com/css?family=Karla:400,700|Rubik:
400,500,700');
```

Since the fonts are a third-party dependency, it's important to import them first.

Introducing your own variables

While Bulma uses a single font family, the new design uses two. You need to create a new variable to store this second family.

After importing the fonts but **before** importing Bulma, add:

```
// New variables
$family-heading: "Rubik", BlinkMacSystemFont, -apple-system, "Helvetica",
"Arial", sans-serif;
```

Until further notice, you will have to add the new Sass snippets just **before** the `@import "node_modules/bulma/bulma";` line.

Rubik will be used as font for headings mainly. The other font families act as a fallback in case Rubik doesn't load.

The design will also heavily use a new type of shadow. Since it's going to be re-used a lot, it's better to store it as a variable too:

```
$large-shadow: 0 10px 20px rgba(#000, 0.05);
```

Understanding Bulma's variables

Bulma comes with three sets of variables:

- **initial variables** are a collection of Sass variables that are assigned a *literal* value like `$blue: hsl(217, 71%, 53%)`
- **derived variables** either reference an initial variable like `$link: $blue`, or use a Sass function to determine their value like `$green-invert: findColorInvert($green)`
- **component variables** are specific to each Bulma element or component, and reference either a previously defined variable, or a new literal

This can create a **chain**. For example:

- In `initial-variables.sass`, the color blue uses a literal value: `$blue: hsl(217, 71%, 53%)`
- In `derived-variables.sass`, the `$link` color uses that shade of blue: `$link: $blue`
- In `breadcrumb.sass`, the breadcrumb items' color use that link color: `$breadcrumb-item-color: $link`

This provides Bulma users with a lot of **flexibility** in terms of customization:

- You can update the `$blue` value and it will be reflected throughout the website
- Or you can set `$link: $green` to update all links, and the breadcrumb items too
- Or you can only choose to update the breadcrumb items to be red instead: `$breadcrumb-item-color: $red`
- Or you can do both: make all links green, but have all breadcrumb items red

The purpose of this setup is to both:

- Make it easy to update a single value everywhere (since `$blue` is defined in a single location)
- Still allow elements and components to be styled **individually**

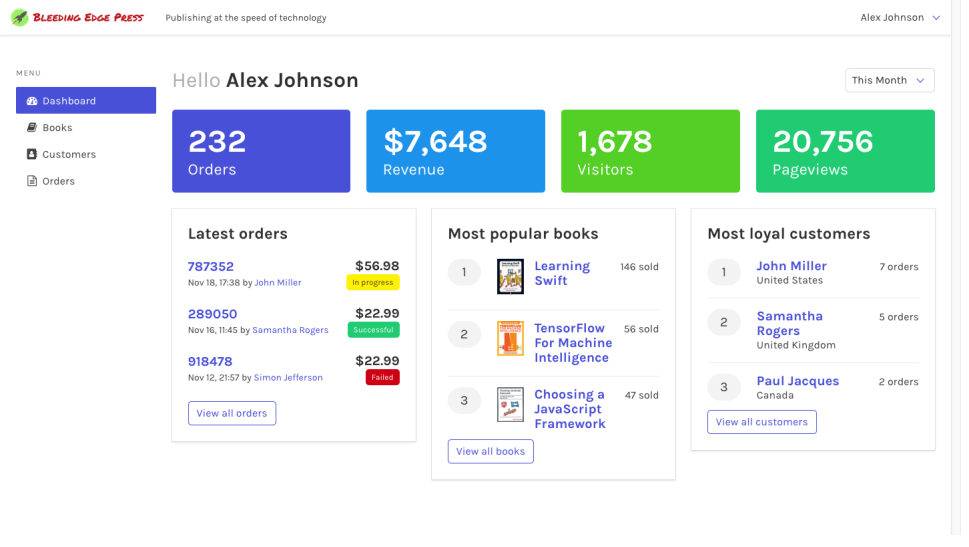
Overriding Bulma's initial variables

The new design comes with new brand colors, a second font Karla, and a bigger border radius. Using this new branding is straightforward: update their respective variables with their new values, and the changes will be reflected throughout the website.

In `custom.scss`, write the following:

```
// Initial variables
$turquoise: #5dd52a;
$red: #D30012;
$yellow: #FFF200;
$green: #24D17D;
$blue: #525adc;

$family-sans-serif: "Karla", BlinkMacSystemFont, -apple-system, "Helvetica",
"Arial", sans-serif;

$radius: 5px;
```

Remember to add this **before** importing Bulma.



The colors have been updated, and the body font is now Karla.

All Bulma variables use the `!default` flag. It's a Sass feature that means a variable's value will be assigned a default value *unless* it has been assigned one before.

That's why importing Bulma *after* having set the new variables still works, and the new brand colors are preserved.

Overriding Bulma's component variables

The new design has a slightly darker page background. This is defined in the `generic.sass`. Instead of writing a new CSS rule, you only need to update the appropriate variable, in this case `$body-background-color`.

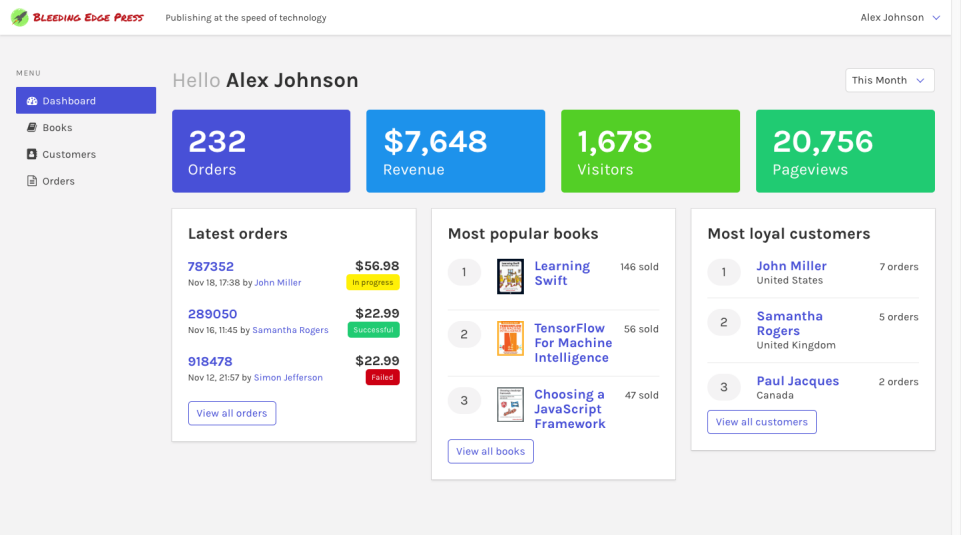
You can use one of Bulma's initial variables: `$white-ter`. To access it, you need to import it:

```
// Import the rest of Bulma's initial variables
@import "node_modules/bulma/sass/utilities/initial-variables";
```

All initial variables are now accessible and can be used to update the component variables.

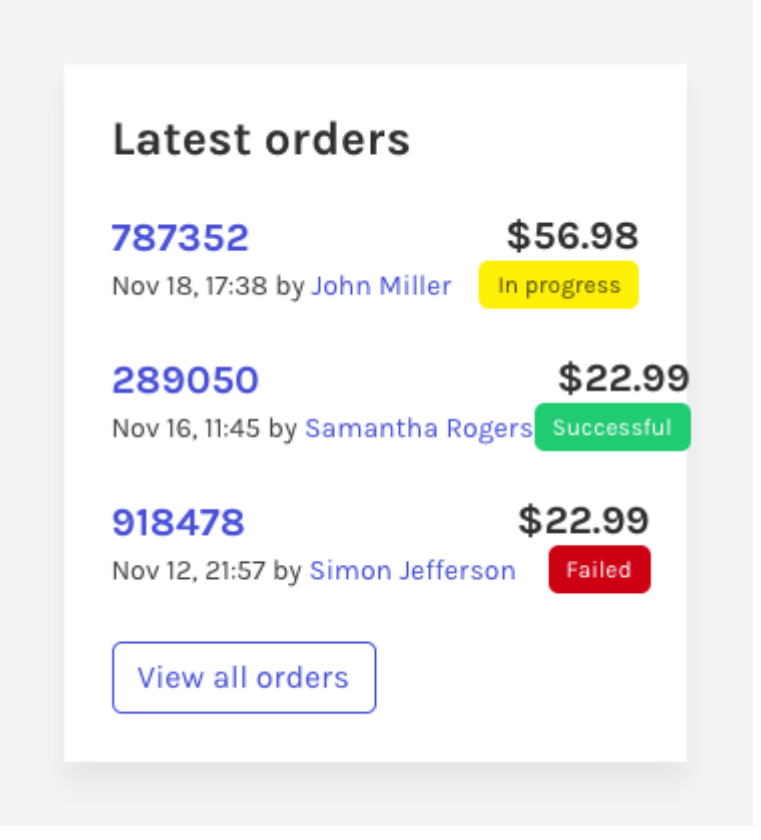
After having imported the initial variables, but before importing the rest of Bulma, reassign this variable:

```
$body-background-color: $white-ter;
```



The next step is to make use of the `$large-shadow` created earlier. Both the Bulma box and the card can use it, and the box also requires a bit more padding:

```
$box-padding: 2rem;
$box-shadow: $large-shadow;

$card-shadow: $large-shadow;
```

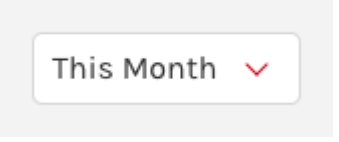


In general, the new design is more spaced out. Increase the gap between columns:

```
$column-gap: 1rem;
```

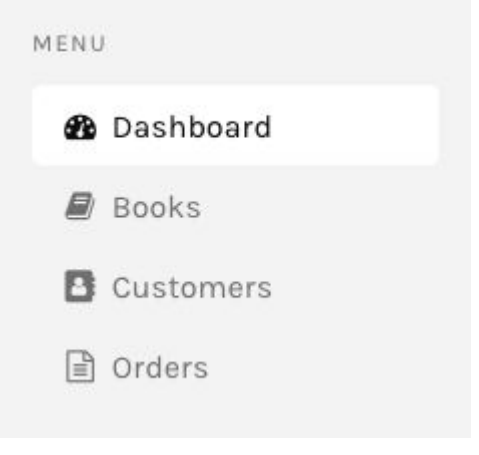
Button and inputs still have blue shadow when focused. A gray one looks better here, combined with a red dropdown arrow:

```
$button-focus-box-shadow-color: rgba($black, 0.1);

$input-arrow: $red;
$input-focus-box-shadow-color: rgba($black, 0.1);
```



The sidebar menu is a bit too prominent. These values will turn it grayscale:

```
$menu-item-color: $grey;
$menu-item-hover-background-color: transparent;
$menu-item-active-background-color: $white;
$menu-item-active-color: $black;
$menu-item-radius: $radius;
```



The navbar and the table also need more breathing space:

```
$navbar-height: 6rem;
$navbar-item-img-max-height: 3rem;
$navbar-item-hover-background-color: transparent;
$navbar-dropdown-border-top: none;

$table-cell-border: 2px solid $white-ter;
$table-cell-padding: 0.75em 1.5em;
```

By simply overriding Bulma's variables, and without writing any CSS, the design has already changed a lot: new color scheme, additional font, and better spacing.

Updating the HTML

The taller navbar is now more prominent. But the logo is too wide:

To save horizontal space, split the icon and the type. Replace the `logo.png` with the `icon.png`:

```
<a class="navbar-item">
  <img src="images/icon.png">
</a>
```

Move the type with the tagline:

```
<div class="navbar-item">
  <div>
    <img src="images/type.png" width="250.5" height="21">
    <br>
    <small>Publishing at the speed of technology</small>
  </div>
</div>
```

Custom rules

Because Bulma is written in Sass, you can use all of the language's features:

- variables
- nesting
- mixins
- extends

You have already used new variables. To customize the design even more, you can use extends and nesting.

All of the code from now on has to be written at the **end** of the file, *after* having imported Bulma.

Second font

Bulma doesn't have a second font family. So you have to write your own CSS rules. Luckily, the classes are easy to extend.

After `@import "node_modules/bulma/bulma";`, write the following:

```
%heading {
  font-family: $family-heading;
  font-weight: 500;
}
```

This is a **Sass placeholder**: this allows you to combine multiple selectors into a single rule.

Bigger controls

The Bulma controls (buttons, inputs, select dropdown, pagination links...) are redefined in this new design: slightly bigger, with no inner shadow or border.

The new control size will be re-used a few times, so it's better to define a new variable:

```
$control-size: 2.75em;
```

A few Bulma elements have to be updated at once:

```
.button,
.input,
.select select,
.pagination-previous,
.pagination-next,
.pagination-link {
  border-width: 0;
  box-shadow: none;
  height: $control-size;
  padding-left: 1em;
  padding-right: 1em;
}
```

The controls with icons and the select dropdown have to accomodate for these bigger controls:

```
.control.has-icons-left {
  .input,
  .select select {
    padding-left: $control-size;
  }

  .icon {
    height: $control-size;
    width: $control-size;
  }
}

.select {
  &:not(.is-multiple) {
    height: $control-size;
  }
}

.select select {
  &:not([multiple]) {
    padding-right: $control-size;
  }
}
```

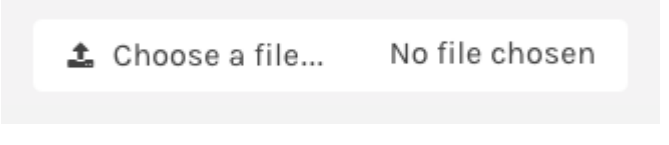
The use of a Sass variable is very useful here. If you change your mind about the `$control-size`, you only need to update the value in one location.

The button borders have been removed, but the outlined buttons still need one:

```
.button {
  &.is-outlined {
    border-width: 2px;
  }
}
```

The last controls to update are the file upload ones:

```
.file-cta,
.file-name {
  background-color: $white;
  border-width: 0;
}
```

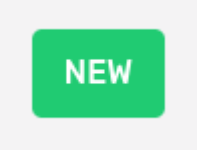


Using the Rubik font

The **Rubik font** is bold and modern, which makes it a perfect contender for titles, labels, and interactive elements like buttons.

Change the button's default background, and make them bolder with uppercase letters:

```
.button {
  @extend %heading;
  background-color: rgba(#000, 0.05);
  text-transform: uppercase;
}
```

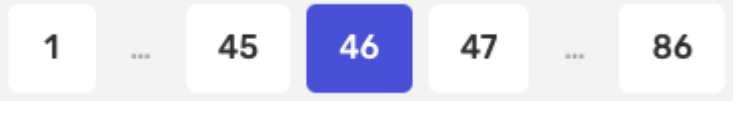


The breadcrumb can follow the same rule:

```
.breadcrumb {
  @extend %heading;
  text-transform: uppercase;
}
```

To make the pagination items look like buttons, use Rubik as well and remove the borders:

```
.pagination {
  @extend %heading;
}

.pagination-previous,
.pagination-next,
.pagination-link {
  background-color: $white;
  border-width: 0;
  min-width: $control-size;
}
```

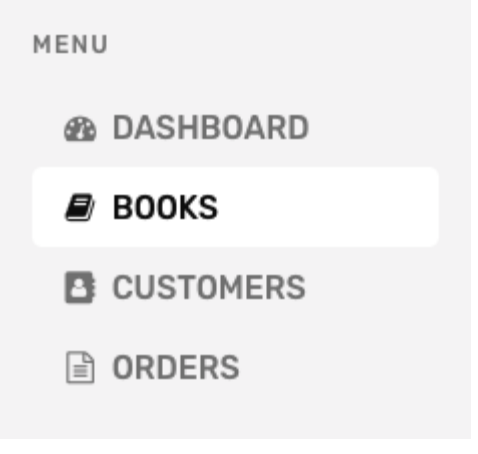


Updating the sidebar menu

The menu's component variables have already been updated to make it grayscale. But the menu still lacks emphasis, and doesn't fit with the rest of the design anymore.

Rubik in uppercase is the solution:

```
.menu {
  @extend %heading;
  text-transform: uppercase;
}
```



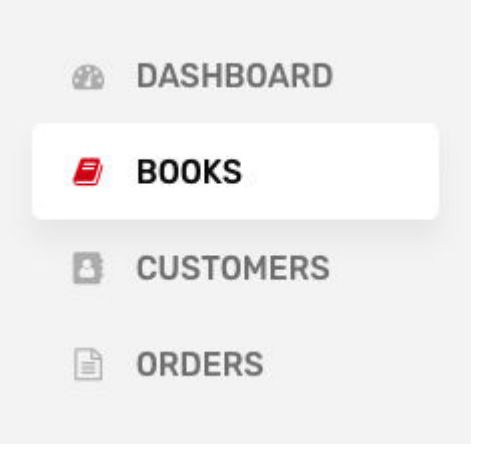
The menu label is not really required anymore. Instead of updating all HTML files, just hide it with CSS:

```
.menu-label {
  display: none;
}
```

By using Sass nesting, you can style the menu list items and its icons very easily:

```
.menu-list {
  a {
    padding: 0.75em 1em;

    .icon {
      color: $grey-light;
      margin-right: 0.5em;
    }

    &.is-active {
```

```
      box-shadow: $large-shadow;

      .icon {
        color: $red;
      }
    }
  }
}
```



The new shadow, while bigger, is still subtle, and gives emphasis to the active menu item.

Fixing the navbar

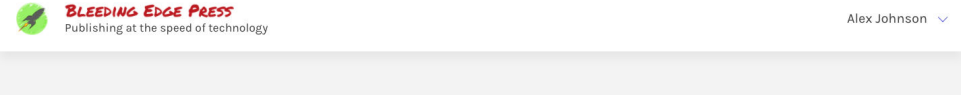
The `$large-shadow` introduced in the beginning has been used throughout the design. The only shadow left is the navbar one. Update it with the new one:

```
.navbar {
  &.has-shadow {
    box-shadow: $large-shadow;
  }
}
```

The navbar has already been customized through Bulma's component variables, but some additional spacing and sizing fixes need to be applied:

```
.navbar-item,
.navbar-link {
  padding: 0.75rem 1.5rem;
```

```
}

.navbar-link {
  padding-right: 2.5em;
}

.navbar-item {
  font-size: $size-5;
}

.navbar-start {
  .navbar-item {
    line-height: 1;
    padding-left: 0;
  }
}
```



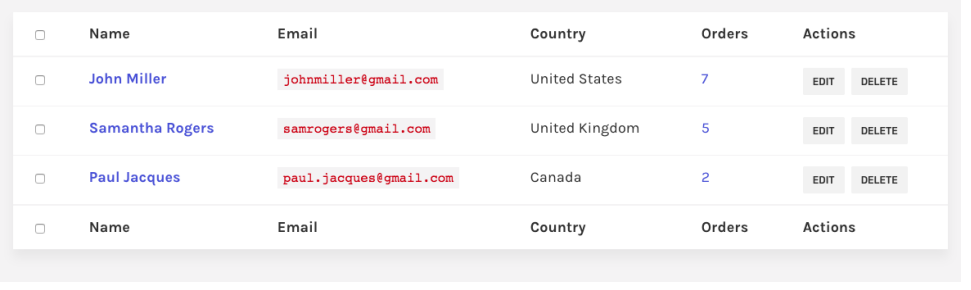
This brings all elements closer together.

Better tables

The table, with its white background, looks flat compared to the rest of the interface, although it's where the primary content lives.

Add a shadow and increase the font size:

```
.table {
  box-shadow: $large-shadow;
  font-size: 1.125rem;
}
```



	Name	Email	Country	Orders	Actions
☐	John Miller	johnmiller@gmail.com	United States	7	EDIT DELETE
☐	Samantha Rogers	samrogers@gmail.com	United Kingdom	5	EDIT DELETE
☐	Paul Jacques	paul.jacques@gmail.com	Canada	2	EDIT DELETE
☐	Name	Email	Country	Orders	Actions

Bold titles

The last elements to update are the titles. Since they tell the user on which page they are, it's better to emphasize them and provide a hierachy in the content:

```
.title {
  @extend %heading;
}

h1.title {
  font-weight: 700;
  text-transform: uppercase;
}
```

Responsiveness with Bulma mixins

The last design fixes required are harder to spot, because they only occur before or after certain breakpoints.

Since Bulma is fully responsive, some of its components are styled according to the viewport size.

Media

The media items in `dashboard.html` combine four elements side by side:

- 2 `media-left`
- 1 `media-content`
- 1 `media-right`

This makes the components squashed on mobile screens.

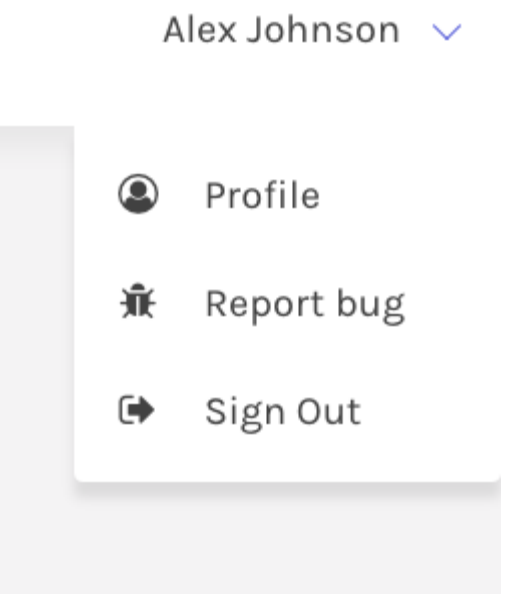
Instead, layout the four elements **vertically**:

```
@include mobile() {
  .media {
    flex-direction: column;
  }

  .media-left {
    margin: 0 0 0.5rem;
  }
}
```

The `mobile()` mixin comes from Bulma itself. It uses the `$mobile` variable breakpoint defined in `initial-variables.sass`. As a result, using this mixin instead of writing your own media query ensures that the responsiveness you write here is **synchronized** with Bulma's own responsive behavior.

The last task is to fix the navbar dropdown on desktop screens:

```
@include desktop() {
  .navbar-dropdown .navbar-item {
    padding: 0.75rem 1.5rem;

    .icon {
      margin-right: 1em;
    }
  }
}
```



The `desktop()` mixin also comes from Bulma and uses the `$desktop` variable.

Final Summary

Thanks to being written in Sass, Bulma is very easy to customize. By overriding a few variables, you can quickly turn the default design into your own branded one.

Lots of developers have used Bulma as a framework to build upon because it comes with sensible defaults that ensure a visually balanced and easy to understand interface. Adding your own personal touch is simply a matter of updating colors, adding some fonts, and tweaking the spacing.

Bulma is also **modular**: with the same setup, instead of importing the rest of Bulma, you can choose to import specific components individually. Each component comes with its own set of variables. **Learn more about modularity**.

Check out the **Bulma Expo** to get inspired.

We hope you have enjoyed this book and are ready to implement Bulma in your own creations!