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

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.

2 Using Bulma with VueJS

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.