FULLSTACK VUE

The Complete Guide to Vue.js

HASSAN DJIRDEH

FULLSTACK.io

Fullstack Vue

The Complete Guide to Vue.js and Friends

Written by Hassan Djirdeh, Nate Murray, and Ari Lerner

Published in San Francisco, California by Fullstack.io.

FULLSTACK.io

Contents

Book Revision

Revision 8 - 08/29/2018

Bug Reports

If you'd like to report any bugs, typos, or suggestions just email us at: **vue@fullstack.io**.

For further help dealing with issues, refer to "How to Get the Most Out of This Book".

Be notified of updates via Twitter

If you'd like to be notified of updates to the book on Twitter, follow us at @fullstackio[1].

We'd love to hear from you!

Did you like the book? Did you find it helpful? We'd love to add your face to our list of testimonials on the website! Email us at: `vue@fullstack.io`.

[1] https://twitter.com/fullstackio

Foreword

Front-end web development has become astoundingly complex. If you've never used a modern JavaScript framework, building your first app that just displays "Hello" can take a whole week! That might sound ridiculous, but most frameworks assume knowledge of the terminal, advanced JavaScript, and tools such as the Node Package Manager (NPM), Babel, Webpack, and sometimes more.

Vue, refreshingly, *doesn't* assume. We call it the "progressive" JavaScript framework because it scales *down* as well as it scales up. If your app is simple, you can use Vue just like jQuery - by dropping in a `<script>` tag. But as your skills and needs grow more advanced, Vue grows with you to make you more powerful and productive.

Chris Fritz - Vue Core Team

Hassan provides a catalyst for that growth in this book. Through project-driven learning, he'll guide you from the simplest examples through the necessary skills for large-scale, enterprise applications. Along the way, you'll learn not only how to solve a variety of problems with Vue, but also the concepts and tools that have become industry standards – no matter what framework you use.

Welcome to the community, have fun, and enjoy the Vue!

😉

– Chris Fritz, @chrisvfritz[2], Vue Core Team

[2] https://twitter.com/chrisvfritz

How to Get the Most Out of This Book

Overview

This book aims to be the **single most useful resource** on learning Vue.js. By the time you're done reading this book, you (and your team) will have everything you need to build reliable, powerful Vue applications.

Vue is built on the premise of simplicity by being designed from the ground up to be incrementally adoptable. After the first few chapters, you'll have a solid understanding of Vue's fundamentals and will be able to build a wide array of rich, interactive web apps with the framework.

But beyond Vue's core, there are tools and libraries that exist in the Vue ecosystem that's often needed to build real-world production scale applications. Things like client-side routing between pages, managing complex state, and heavy API interaction at scale.

This book can be broken down into two parts.

In Part I, we cover all the fundamentals with a progressive, example-driven approach. You'll first **introduce Vue through a Content Delivery Network (CDN)** before moving towards building within Webpack bundled applications. You'll gain a grasp of **handling user interaction**, **working with single-file components**, **understanding simple state management**, and **how custom events work**.

We bookend the first part by introducing Vuex and how Vuex is integrated to manage overall application data architecture.

Part II of this book moves into more **advanced concepts** that you'll often see used in large, production applications. We'll **integrate Vuex to a server-persisted app**, **manage rich forms**, build a multi-page app that uses **client-side routing**, and finally explore how **unit tests** can be written with Vue's official unit testing library.

First, know that you do not need to read this book linearly from cover-to-cover. **However**, we've ordered the contents of the book in a way we feel fits the order you should learn the concepts. Some sections in Part II assume you've acquired certain fundamental concepts from Part I. As a result, we encourage you to learn all the concepts in Part I of the book first before diving into concepts in Part II.

Second, keep in mind this package is more than just a book - it's a course complete with example code for every chapter. Below, we'll tell you:

- how to approach **the code examples** and
- **how to get help** if something goes wrong

Vue 2.x

In Sept. 2016[3], the Vue framework was rewritten and released as version 2.0. Vue 2.0 introduced new concepts such as the Virual DOM, render functions, and server-side rendering capabilities. In addition, version 2.0 was rewritten to provide significant performance improvements over v1.

This book covers, and will always cover, the latest release of Vue - **which is currently labelled as version 2.x**.

Running Code Examples

This book comes with a library of runnable code examples. The code is available to download from the same place where you purchased this book.

If you have any trouble finding or downloading the code examples, email us at **vue@fullstack.io**.

Webpack projects

For Webpack-bundled projects, we use the program `npm`[4] to run examples. You can install the application dependencies with:

```
npm install
```

And boot apps with one of the following commands:

```
npm run start
```

or

```
npm run serve
```

With every chapter, we'll reiterate the commands necessary to install application dependancies and boot example code.

After running `npm run start` or `npm run serve`, you will see some output on your screen that will tell you what URL to open to view your app.

> If you're unfamiliar with `npm`, we cover how to get it installed in the "Setting Up" section in the second chapter.

If you're ever unclear on how to run a particular sample app, checkout the `README.md` in that project's directory. Every Webpack bundled sample project contains a `README.md` that will give you the instructions you need to run each app.

[3] https://medium.com/the-vue-point/vue-2-0-is-here-ef1f26acf4b8
[4] https://www.npmjs.com/

Direct `<script>` Include

For simpler examples, we've resorted to directly including the Vue library from a Content Delivery Network (CDN) to get the app up and running as fast as possible.

In this book, applications that introduce Vue from a CDN often consist of a single HTML file (`index.html`) for markup and a single JS file (`main.js`) for all Vue code. With these examples, we'll be able to run the app by opening the `index.html` file in our browser (e.g. right click `index.html` file and select `Open With > Google Chrome`). Since the Vue library is hosted externally in these cases, **these examples will require your machine to be connected to the internet**.

Code Blocks and Context

The majority of code blocks in this book is pulled from a **runnable code example** which you can find in the sample code. For example, here is a code block pulled from the first chapter:

upvote/app_5/main.js

```
new Vue({
  el: '#app',
  data: {
    submissions: Seed.submissions
  },
  computed: {
    sortedSubmissions () {
      return this.submissions.sort((a, b) => {
        return b.votes - a.votes
      });
    }
  },
  components: {
    'submission-component': submissionComponent
  }
});
```

Notice that the header of this code block states the path to the file which contains this code: `upvote/app_5/main.js`.

Certain code examples will resemble building blocks to get to a certain point and thus may not reflect a code block directly from the sample code. If you ever feel like you're missing the context for a code example, open up the full code file using your favorite text editor. **This book is written with the expectation that you'll also be looking at the example code alongside the manuscript**.

For example, we often need to `import` libraries to get our code to run. In the early chapters of the book we show these `import` statements, because it's not clear where the libraries are coming from

otherwise. However, the later chapters of the book are more advanced and they focus on *key concepts* instead of repeating boilerplate code that was covered earlier in the book. **If at any point you're not clear on the context, open up the code example on disk**.

Code Block Numbering

In this book, we mostly build larger examples in steps. If you see a file being loaded that has a numeric suffix, that generally means we're building up to something bigger.

For instance, the code block above has the file path: `upvote/app_5/main.js`. When you see the `-N.js` syntax, that means we're building up to a final version of the file. You can jump into that file and see the state of all the code at that particular stage.

Instruction for Windows users

> All the code in this book has been tested on a Windows machine. If you have any issues running the code on Windows, send us an email[5] and we'll try to help you get it resolved.

Ensure Node.js and npm are installed

If you're on a Windows machine and have yet to do any web development on it, you can install the Node.js Windows Installer from the Node.js[6] website. With Node.js (and npm) appropriately installed, you should be able to start the Webpack-bundled Node.js projects in the book as expected.

See this tutorial[7] for installing Node.js and npm on Windows.

Getting Help

While we've made every effort to be clear, precise, and accurate you may find that when you're writing your code you run into a problem.

Generally, there are three types of problems:

- A "bug" in the book (e.g. how we describe something is wrong)
- A "bug" in our code
- A "bug" in your code

[5] vue@fullstack.io
[6] http://nodejs.org
[7] http://blog.teamtreehouse.com/install-node-js-npm-windows

If you find an inaccuracy in how we describe something, or you feel a concept isn't clear, email us! We want to make sure that the book is both accurate and clear.

Similarly, if you've found a bug in our code we definitely want to hear about it.

If you're having trouble getting your own app working (and it isn't *our* example code), this case is a bit harder for us to handle. If you're still stuck, we'd still love to hear from you.

Emailing Us

If you're emailing us asking for technical help, here's what we'd like to know:

- What revision of the book are you referring to?
- What operating system are you on? (e.g. Mac OS X 10.13.2, Windows 95)
- Which chapter and which example project are you on?
- What were you trying to accomplish?
- What have you tried already?
- What output did you expect?
- What actually happened? (Including relevant log output.)

The **absolute best way to get technical support** is to send us a short, self-contained example of the problem. Our preferred way to receive this would be for you to send us a Plunkr link by using this URL[8] as a template.

That URL contains a runnable, boilerplate Vue app. If you can copy and paste your code into that project, reproduce your error, and send it to us **you'll greatly increase the likelihood of a prompt, helpful response**.

When you've written down these things, email us at **vue@fullstack.io**. We look forward to hearing from you.

Get excited!

Writing web apps with Vue is *fun*. And by using this book, **you're going to learn how to build real Vue apps** fast. (Much faster than spending hours parsing out-dated blog posts.)

If you've written client-side JavaScript before or used existing JavaScript frameworks, you'll find Vue refreshingly intuitive. If this is your first serious foray into the front-end, you'll be *blown away* at how quickly you can create something worth sharing.

So hold on tight - you're about to become really proficient with Vue, and have a lot of fun along the way. Let's dig in!

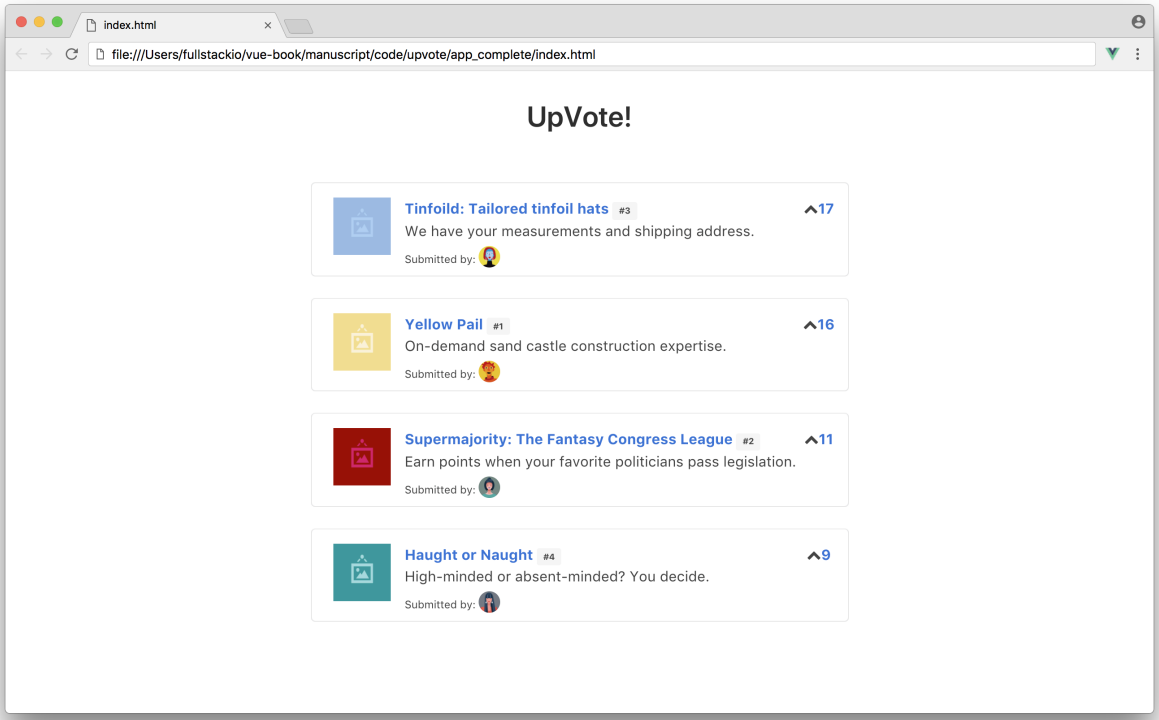
- Hassan (@djirdehh[9]), Nate, and Ari

[8] http://bit.ly/2orvQcd
[9] https://twitter.com/djirdehh

I - Your first Vue.js Web Application

Building UpVote!

On our first step to learning Vue, we're going to build a simple voting application (named UpVote!) that takes inspiration from popular social feed websites like Reddit[10] and Hacker News[11].



Completed version of the app

UpVote! focuses on displaying a list of submissions that users can vote on. Each submission will present some information about itself like an image, title, and description. All submissions are sorted instantaneously by number of votes. The up-vote icon in each submission will allow users to increase vote numbers and subsequently rearrange submission layout.

With UpVote!, we'll become familiar with how Vue approaches front-end development by understanding the basic fundamentals associated with the library. By the end of the chapter we'll be well on our way to building dynamic front-end interfaces thanks to Vue's simplicity!

[10] https://reddit.com
[11] https://hackernews.com

Development environment setup

Code editor

Regardless of experience, whenever developing for the web, we'll need a code editor to write our application (this is true for all code in this book). It's most important to be comfortable with your code editor, so if you have one you like, stick with it. If not, we recommend Atom[12], Sublime Text 3[13], or Visual Studio Code[14].

Development Environment

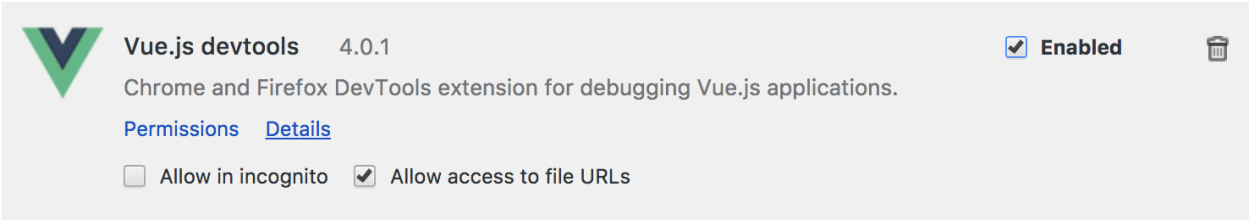
For this chapter, we'll focus on getting our Vue app up and running as fast as possible, so we'll simply introduce Vue through a Content Delivery Network (CDN). We'll take a deeper look into installing all the necessary libraries for our development environment in the next chapter.

Browser

We highly recommend using the Google Chrome Web Browser[15] to develop Vue apps since we'll be using the Chrome developer toolkit[16] throughout this book. To follow along with our development and debugging, we recommend installing Chrome, if not installed already.

With Chrome, Vue provides an incredibly useful extension, Vue.js devtools[17] that simplifies debugging of Vue applications. We'll be using the devtools at separate points throughout the book so we encourage you to install it as well.

Note: With certain chapters in this book (like this chapter for example), we'll be working with applications opened via `file://` protocol. To make the Vue devtools work for these pages, you'll need to check "Allow access to file URLs" for the extension in Chrome's extension manager:



Allow access to file URLs

[12] http://atom.io
[13] https://www.sublimetext.com/
[14] https://code.visualstudio.com/
[15] https://www.google.com/chrome/
[16] https://developers.google.com/web/tools/chrome-devtools/
[17] https://github.com/vuejs/vue-devtools

JavaScript ES6/ES7

JavaScript is the language of the web. It runs on many different browsers, including Google Chrome, Firefox, Safari, Microsoft Edge, and Internet Explorer. Different browsers have different JavaScript interpreters which execute JavaScript code.

Its widespread adoption as the Internet's client-side scripting language led to the formation of a standards body which manages its specification. The specification is called **ECMAScript** or ES.

The 5th edition of the specification is called ES5. We think of ES5 as a "version" of the JavaScript programming language. Finalized in 2009, ES5 was adopted by all major browsers within a few years.

The 6th edition of JavaScript is referred to as ES6. Finalized in 2015, the latest versions of major browsers are still finishing adding support for ES6 as of 2017. ES6 provides a significant update. It contains a whole host of new features for JavaScript, almost two dozen in total. JavaScript written in ES6 is tangibly different than JavaScript written in ES5.

ES7, a much smaller update that builds on ES6, was ratified in June 2016. ES7 contains only two new features.

To take advantage of the future versions of JavaScript, we want to write our code in ES6/ES7 today. We'll also want our JavaScript to run on older browsers until they fade out of widespread use.

This book is written using the JavaScript ES7 version. As ES6 ratified a majority of these new features, we'll commonly refer to these new features as ES6 features.

ES6 is sometimes referred to as ES2015, the year of its finalization. ES7, in turn, is often referred to as ES2016.

Getting started

Sample Code

All the code examples/snippets contained in this chapter (and all the other chapters) are available in the code package that came with the book. In the code package we'll see completed versions of the apps as well as boilerplates to help us get started. Each chapter provides detailed instruction on how to follow along on our own.

While coding along with the book is not necessary, we highly recommend doing so. Playing around with examples and sample code will help solidify and strengthen understanding of new concepts.

Previewing the application

We'll begin this chapter by taking a look at a working implementation of the UpVote! app.

Let's open up the sample code that came with the book and locate the `upvote/` folder with our machines file navigator (Finder for OS X or Windows Explorer on Windows) or through our code editor (e.g. Sublime). By opening the `upvote/` folder, we'll see all the sub-directories contained within the sample app:

```
upvote
    app/
    app_1/
    app_2/
    app_3/
    app_4/
    app_5/
    app_complete/
    public/
```

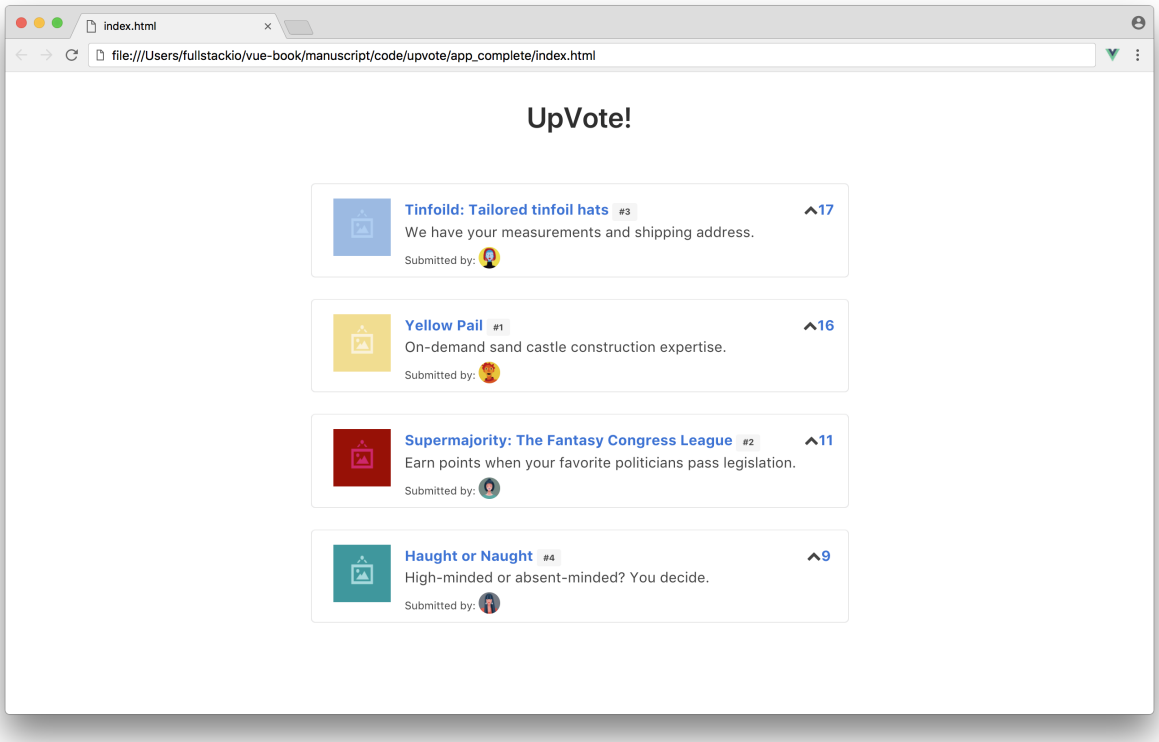
We've included each version of the app as we build it up throughout this chapter (`app_1/`, `app_2/`, etc). Each code block in this chapter references which app version it is contained within. We can copy and paste longer code insertions from these app versions into our local `app/` folder, the starting point of our application.

The `public/` sub-folder hosts all the images and custom styles we'll use within our application.

`app_complete` represents the completed state of our application. Opening the `app_complete` folder, we'll see there are just three files located inside:

```
app_complete
    index.html
    main.js
    seed.js
```

We can see the running application by right clicking on the `index.html` file and selecting `Open With > Google Chrome`.



Completed version of the app

Notice how the submissions are all sorted from highest to lowest number of votes. The application will keep the posts sorted by number of votes, moving them around as the votes change *without* reloading the page.

Prepare the app

Let's begin building the application. We're going to be working entirely from the `app/` directory. By opening the files within `app/` in a text editor, we'll see some boilerplate code contained in the `index.html` and `seed.js` files, while `main.js` is left blank.

Let's begin by looking inside the `index.html` file:

upvote/app/index.html

```
<!DOCTYPE html>
<html>

<head>
  <link rel="stylesheet"
    href="https://cdnjs.cloudflare.com/ajax/libs/bulma/0.5.3/css/bulma.css">
  <link rel="stylesheet"
    href="https://use.fontawesome.com/releases/v5.0.6/css/all.css">
  <link rel="stylesheet"
    href="../public/styles.css" />
</head>

<body>
  <div id="app">
    <h2 class="title has-text-centered dividing-header">UpVote!</h2>
  </div>

  <script src="https://unpkg.com/vue"></script>
  <script src="./seed.js"></script>
  <script src="./main.js"></script>
</body>

</html>
```

In our `<head>` tag, there are three stylesheet dependancies we've included in our application:

upvote/app/index.html

```
<head>
  <link rel="stylesheet"
    href="https://cdnjs.cloudflare.com/ajax/libs/bulma/0.5.3/css/bulma.css">
  <link rel="stylesheet"
    href="https://use.fontawesome.com/releases/v5.0.6/css/all.css">
  <link rel="stylesheet"
    href="../public/styles.css" />
</head>
```

We've introduced Bulma[18] as our applications CSS framework, Font Awesome[19] for icons, and our own `styles.css` file that lives in our `public` folder.

[18] http://bulma.io/
[19] http://fontawesome.io/

For this project, we're using Bulma[20] for styling.

Bulma is a CSS framework, much like Twitter's popular Bootstrap[21] framework. It provides us with a grid system and some simple styling. We don't need to know Bulma in-depth in order to go through this chapter (or this book).

We'll always provide all the styling code that is needed. At some point, it's a good idea to check out the Bulma docs[22] to get familiar with the framework and explore how to use it in other projects we'll build in the future.

The heart of our application lives in the few lines within our `<body>` tag which currently looks like this:

upvote/app/index.html

```
<div id="app">
  <h2 class="title has-text-centered dividing-header">UpVote!</h2>
</div>
```

The `class` attributes refer to CSS styles and are safe to ignore in the context of our application. Not paying attention to those, we can see we have a title for the page (`h2`) and a `<div>` element with an `id` of `app`.

The `<div>` element with the `id` of `app` is where our Vue application will be loaded and *attached* onto the template. In other words, our Vue application will be **mounted** on to this particular element.

The next few lines tells the browser which JavaScript files to load:

upvote/app/index.html

```
<script src="https://unpkg.com/vue"></script>
<script src="./seed.js"></script>
<script src="./main.js"></script>
```

The first `<script>` tag loads the latest version of Vue from a Content Delivery Network (CDN) at unpkg[23]. Using the CDN to load the Vue dependency is the simplest and quickest way to introduce Vue to an application.

A Content Delivery Network (CDN) is a system of services that deliver content to users based on their geographical location and the content delivery server. Using CDN's have the benefit of decreasing server load and providing faster loading times to users who've already downloaded the content.

Most CDNs are used to deliver static content like common JavaScript libraries, fonts, CSS files, etc. We've also introduced Bulma and Font Awesome through CDNs in our `<head>` tag.

[20] http://bulma.io/
[21] http://getbootstrap.com/
[22] http://bulma.io/documentation/overview/start/
[23] https://unpkg.com

The other two `<script>` tags reference the internal JavaScript files we'll write in the `./seed.js` and `./main.js` files.

Setting up the view

Now that we have a good understanding of our boilerplate code, we can start diving in and writing some code. Let's first set up a template for how a single submission would look like. We'll adapt Bulma's media object[24] as it represents a good starting point.

In our `index.html` we'll insert the following template block right below our `h2` title:

upvote/app_1/index.html

```
    <div class="section">
      <article class="media">
        <figure class="media-left">
          <img class="image is-64x64"
            src="../public/images/submissions/image-yellow.png">
        </figure>
        <div class="media-content">
          <div class="content">
            <p>
              <strong>
                <a href="#" class="has-text-info">Yellow Pail</a>
                <span class="tag is-small">#4</span>
              </strong>
              <br>
                On-demand sand castle construction expertise.
              <br>
              <small class="is-size-7">
                Submitted by:
                <img class="image is-24x24"
                  src="../public/images/avatars/daniel.jpg">
              </small>
            </p>
          </div>
        </div>
        <div class="media-right">
          <span class="icon is-small">
            <i class="fa fa-chevron-up"></i>
            <strong class="has-text-info">10</strong>
          </span>
```

[24] http://bulma.io/documentation/layout/media-object/

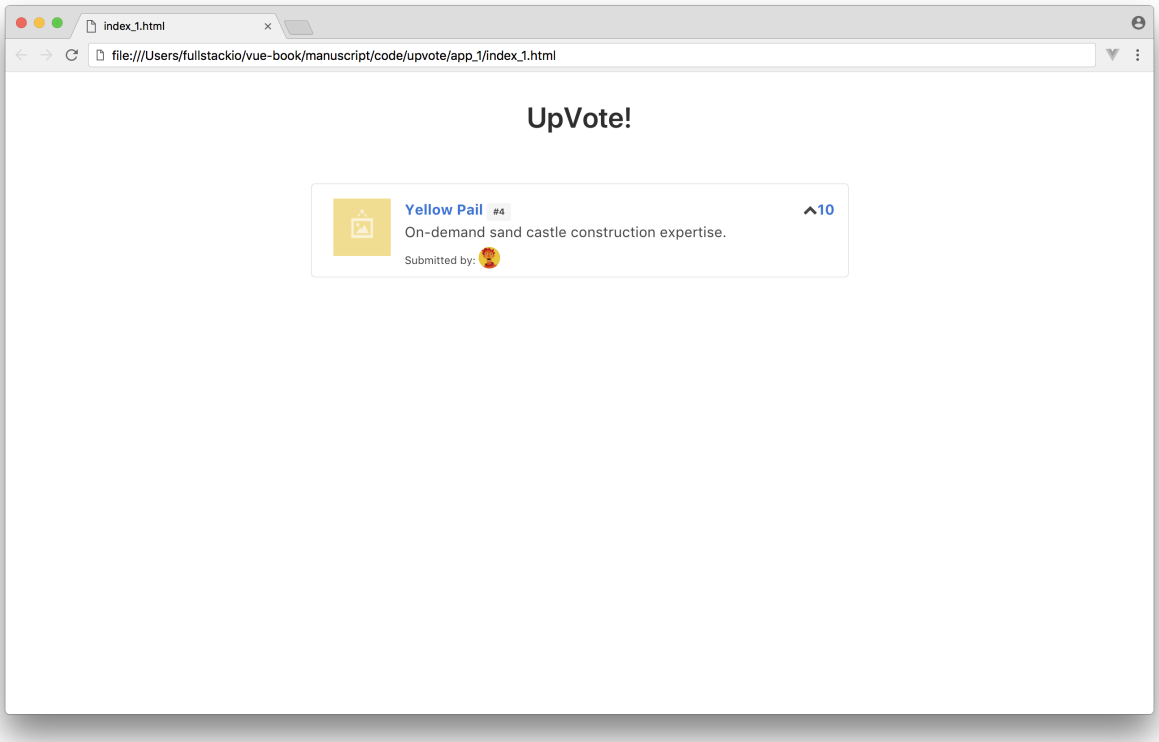
```
      </div>
    </article>
  </div>
```

This template is a slight modification of Bulma's media object[25].

We've added an encompassing `<div>` element over an `<article>` template block. The `<article>` template block is the view for a single submission and has three child DOM elements:

- `<figure>` with `media-left` class which will display the main image of the submission and is positioned to the left.
- `<div>` with `media-content` class which displays the additional details of the submission such as the title, id, description, and avatar of the submitted user.
- `<div>` with `media-right` class which shows a `fa-chevron-up` icon alongside the submission's number of votes.

If we open our `app/index.html` in our Chrome Browser (right click and select `Open With > Google Chrome`), we will see our newly built submission.



A single submission

[25] http://bulma.io/documentation/layout/media-object/

Awesome. We won't write much more HTML markup than what we've just added.

While neat, at the moment our view is static. We've simply hard-coded the title, description and other details. To use this template in a meanigful way, we'll want to change it to be *reactive* (i.e. dynamically data-driven).

Making the view data-driven

Driving the template with data will allow us to dynamically render the view based upon the data that we give it. Let's familiarize ourselves with the applications data model.

The data model

Within our `app` directory, we've included a file called `seed.js`. `seed.js` contains sample data for a list of submissions (it *seeds* our application with data). The `seed.js` file contains a JavaScript object called `Seed.submissions`. `Seed.submissions` is an array of JavaScript objects where each represents a sample submission object:

```
const submissions = [
  {
    id: 1,
    title: 'Yellow Pail',
    description: 'On-demand sand castle construction expertise.',
    url: '#',
    votes: 16,
    avatar: '../public/images/avatars/daniel.jpg',
    submissionImage: '../public/images/submissions/image-yellow.png',
  },
  // ...
]
```

Each submission has a unique `id` and a series of properties including `title`, `description`, `votes`, etc.

Since we only have a single submission displayed in our view, we'll first focus on getting the data from a single submission object (i.e. `submissions[0]`) on to the template.

The Vue Instance

The Vue instance is the starting point of all Vue applications. A Vue instance accepts an **options** object which can contain details of the instance such as its template, data, methods, etc. Root level Vue instances allow us to reference the DOM with which the instance is to be mounted/attached to.

Let's see an example of this by setting up the Vue instance for our application. We'll write all our Vue code for the rest of this chapter inside the `main.js` file. Let's open `main.js` and create the Vue instance using the `Vue` function:

```
new Vue({
  el: '#app'
});
```

We've just specified the HTML element with the id of `app` to be the mounting point of our Vue application, by using the `el` option and providing it a string value of `#app`. Anywhere within this element can Vue JavaScript code now be used.

The Vue instance can also return data that needs to be handled within the view. This data must be specified within a `data` object in the instance. This is how we'll arrange the connection between the data in our `seed.js` file and the template view.

Let's update the instance by specifying a new `data` object. In the object, we'll include a `submissions` key that will have the same value as the `Seed.submissions` array:

upvote/app_2/main.js

```
new Vue({
  el: '#app',
  data: {
    submissions: Seed.submissions
  }
});
```

In the HTML template, we can now reference all submission data by accessing `submissions`.

The `Vue` object constructor is available on the global scope since we've included the `<script />` tag, that loads Vue, in our `index.html` file. Without including this tag, the `Vue` function won't be available and we'll be presented with a console error stating `Uncaught ReferenceError: Vue is not defined`.

With our Vue instance created and containing submission data, we can now work towards synchronizing data in the model to the view. In other words, we can now **data bind** the instance's data to the DOM.

Data binding

The simplest form of data binding in Vue is using the 'Mustache' syntax which is denoted by double curly braces `{{}}`. We'll apply this syntax to bind all the text within our HTML (e.g. title, description, etc.).

The 'Mustache' syntax however cannot be used in HTML attributes like `href`, `id`, `src` etc. Vue provides the native **`v-bind`** attribute (this is known as a Vue directive) to bind HTML attributes. We'll use this directive to update the `src` attributes in our template.

The Vue syntax may take some brief time to get used to, both within template manipulation as well as on the JavaScript side.

We'll gain familiarity on syntax/semantics as we continue to write code within this book.

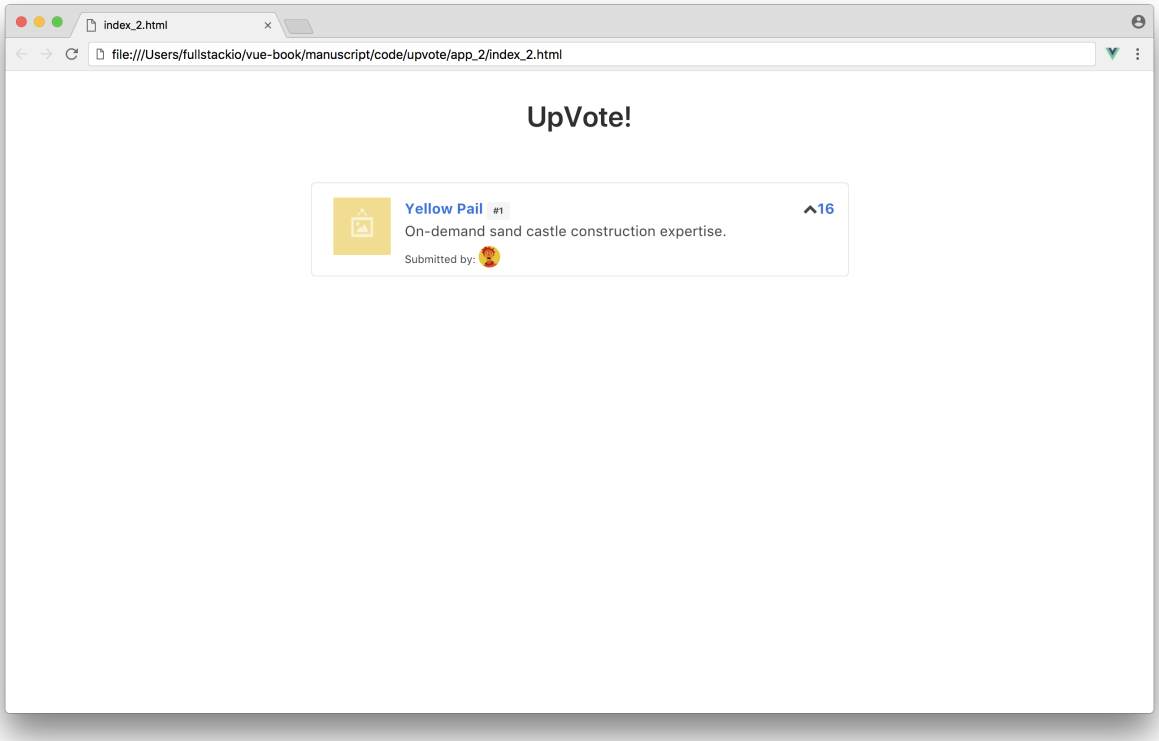
Let's swap the hard-coded data in the template to now reference the content in the first object in the `submissions` array, `submissions[0]`. This will make the newly added template block now look like this:

upvote/app_2/index.html

```
<div class="section">
  <article class="media">
    <figure class="media-left">
      <img class="image is-64x64"
        v-bind:src="submissions[0].submissionImage">
    </figure>
    <div class="media-content">
      <div class="content">
        <p>
          <strong>
            <a v-bind:href="submissions[0].url" class="has-text-info">
              {{ submissions[0].title }}
            </a>
            <span class="tag is-small">#{{ submissions[0].id }}</span>
          </strong>
          <br>
            {{ submissions[0].description }}
          <br>
          <small class="is-size-7">
            Submitted by:
            <img class="image is-24x24"
              v-bind:src="submissions[0].avatar">
          </small>
        </p>
      </div>
    </div>
    <div class="media-right">
      <span class="icon is-small">
        <i class="fa fa-chevron-up"></i>
        <strong class="has-text-info">{{ submissions[0].votes }}</strong>
      </span>
    </div>
  </article>
</div>
```

If we've bound everything appropriately, we should see no change in our view (since the hard-coded information was the same content in our `submissions[0]` object).

Let's refresh our browser and see our template be rendered again.



Submission with bound data

List rendering

We've successfully created our Vue instance and **bound** a single submission object in our view. Our next objective is to render all the submission objects to our view by displaying each submission object as a separate template block.

Since we're going to be rendering a *list* of submission objects, we're going to use Vue's native **`v-for`** directive.

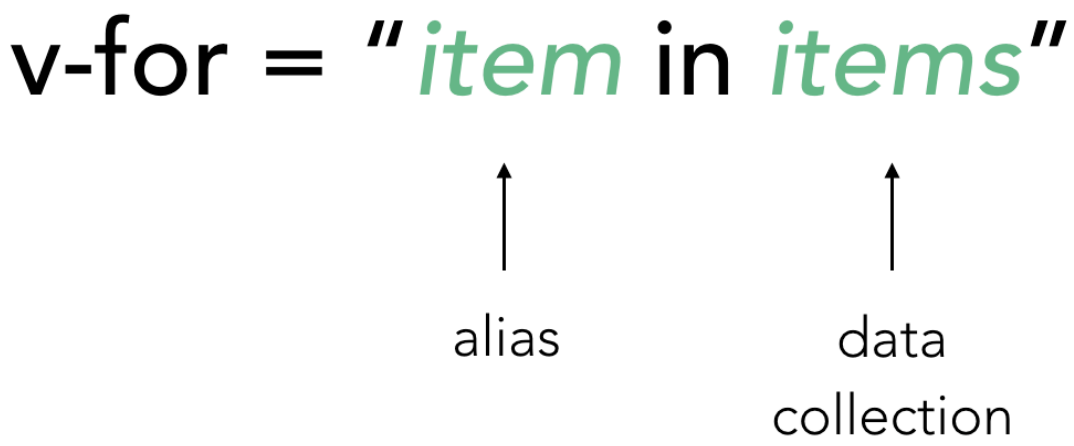
`v-for` directive

The `v-for` directive is used to render a list of items based on a data source. In our case, we would like to render a submission post for each of the submission objects in our `Seeds.submission` array.

The `<article>` element in the `index.html` file, which is a standard HTML element, currently displays a single submission post:

```
<article class="media">
  <!-- Rest of the submission template -->
</article>
```

The `v-for` directive requires a specific syntax along the lines of `item in items`, where `items` is a data collection and `item` is an alias for every element that is being iterated upon:



In our template, since `submissions` is the data collection we'll be iterating over; `submission` would be an appropriate alias to use. We'll add the `v-for` statement to the `<article>` block like this:

```
<article v-for="submission in submissions" class="media">
  <!-- Rest of the submission template -->
</article>
```

key

It's common practice to specify a `key` attribute for every iterated element within a rendered `v-for` list. Vue uses the `key` attribute to create unique bindings for each node's identity.

To specify this uniqueness to each item in the list, we'll assign a `key` to every iterated submission. We'll use the `id` of a submission since a submission's `id` would never be equal to that of another submission. Because we're using dynamic values, we'll need to use `v-bind` to bind our key to the `submission.id`:

```
<article v-for="submission in submissions" v-bind:key="submission.id"
  class="media">
  <!-- Rest of the submission template -->
</article>
```

If there were any dynamic changes made to a `v-for` list *without* the `key` attribute, Vue will opt towards changing data within each element *instead* of moving the DOM elements accordingly. By specifying a unique `key` attribute to each iterated item, we're now telling Vue to reorder elements if needed.

The Vue docs[26] explains the importance of the `key` attribute in more detail.

In our template, let's now change the `submissions[0]` references and update it to use the iterated array instance variable `submission`:

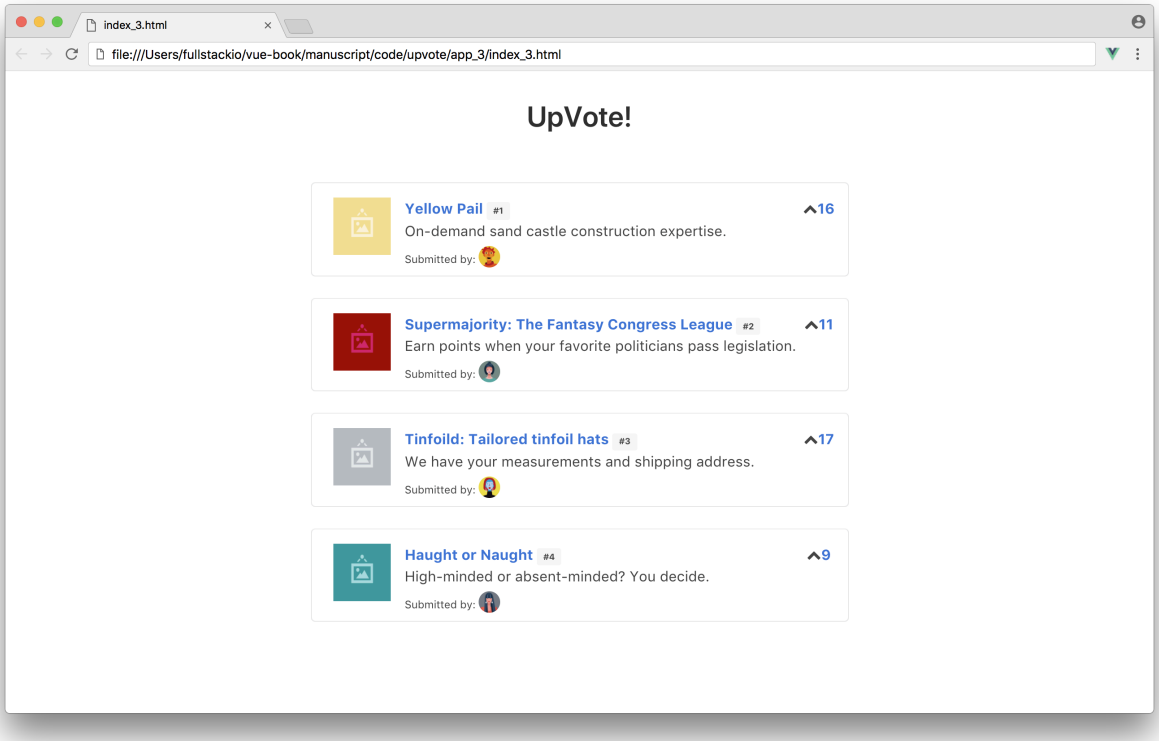
upvote/app_3/index.html

```
    <div class="section">
      <article v-for="submission in submissions" v-bind:key="submission.id"
        class="media">
        <figure class="media-left">
          <img class="image is-64x64"
            v-bind:src="submission.submissionImage">
        </figure>
        <div class="media-content">
          <div class="content">
            <p>
              <strong>
                <a v-bind:href="submission.url" class="has-text-info">
                  {{ submission.title }}
                </a>
                <span class="tag is-small">#{{ submission.id }}</span>
              </strong>
              <br>
                {{ submission.description }}
              <br>
              <small class="is-size-7">
                Submitted by:
                <img class="image is-24x24"
                  v-bind:src="submission.avatar">
              </small>
```

[26] https://vuejs.org/v2/guide/list.html#key

```
          </p>
        </div>
      </div>
      <div class="media-right">
        <span class="icon is-small">
          <i class="fa fa-chevron-up"></i>
          <strong class="has-text-info">{{ submission.votes }}</strong>
        </span>
      </div>
    </article>
  </div>
```

Refreshing our browser, we should now expect to see a list of submissions. This is due to the `v-for` directive now dynamically creating a submission `<article>` element for each submission in the seed file.



List of submissions

Sorting

In traditional social feeds (like Reddit[27] and Hacker News[28]), we often see number of votes as the measuring stick that controls the position of different submission posts. Submissions with the highest number of votes appear at the top of the web page with lower voted submissions being positioned at the bottom.

If we go back to our `v-for` statement in the template, we see an iteration of `submission in submissions`. `submissions` is the standard data object that is being used in our view, retrieved from our data source.

Wouldn't it be great if we can somehow specify an iteration like `submission in sortedSubmissions` where `sortedSubmissions` returns a *sorted* array of `submissions` all the time? This is where Vue's `computed` properties come in.

Computed properties

Computed properties are used to handle complex calculations of information that need to be displayed in the view. Below the `data` property in our Vue instance (back in the `main.js` file), we'll introduce a `computed` property `sortedSubmissions` that returns a sorted array of `submissions`:

```
new Vue({
  el: '#app',
  data: {
    submissions: Seed.submissions
  },
  computed: {
    sortedSubmissions () {
      return this.submissions.sort((a, b) => {
        return b.votes - a.votes
      });
    }
  }
});
```

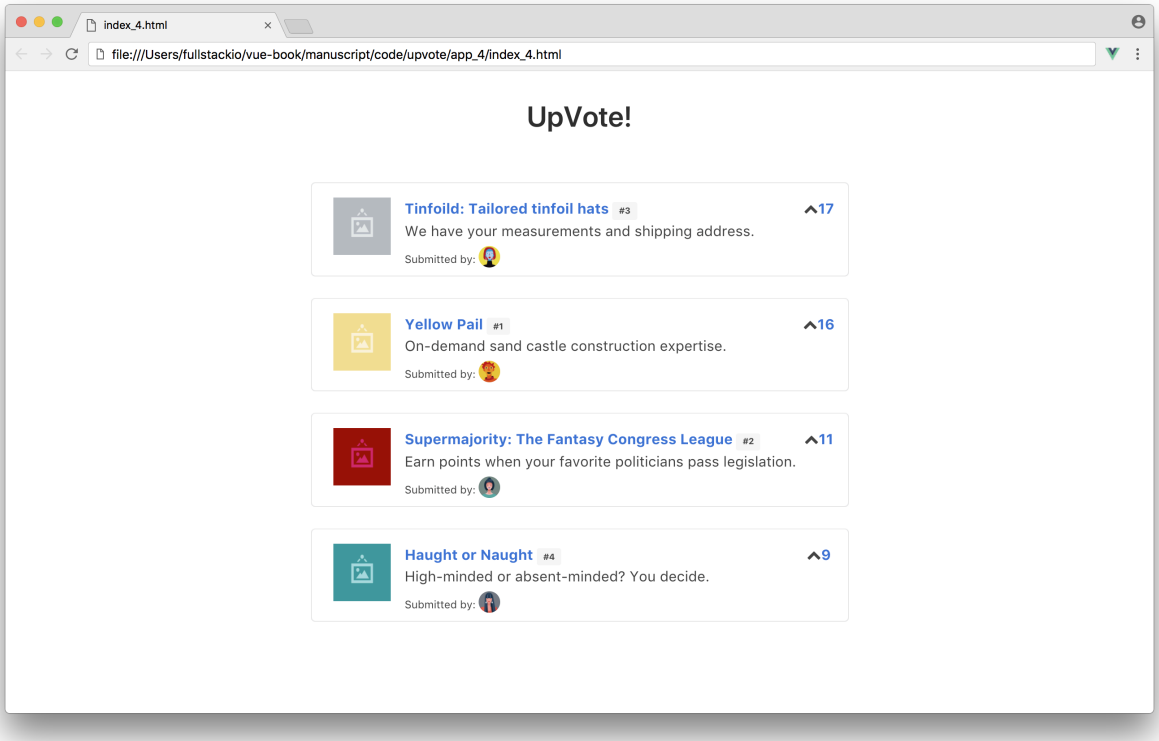
Within a Vue instance, we're able to reference the instance's `data` object with `this`. Hence `this.submissions` refers to the `submissions` object we've specified in our instance's `data`. For sorting we're simply using the native Array object's sort method[29].

In our template where we have our `v-for` expression, we can now replace `submissions` with `sortedSubmissions` as the array to iterate over.

[27] https://reddit.com
[28] https://hackernews.com
[29] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort

```
<article v-for="submission in sortedSubmissions" v-bind:key="submission.id"
  class="media">
  <!-- Rest of the submission template -->
</article>
```

Refreshing our browser, we see the same list of submissions but now appropriately sorted by the number of votes!



Sorted list of submissions

Event handling (our app's first interaction)

When the up-vote icon on each one of the submissions is clicked, we expect it to increase the `votes` attribute for that submission by one. To handle this interaction, we'll be using Vue's native **`v-on`** directive.

The `v-on` directive

The `v-on` directive is used to create event listeners within the DOM.

As all web browsers are event driven, we'll use these events to trigger interaction in our Vue application. For instance, in native JavaScript (i.e. without Vue), we can attach an event listener to a DOM object using the `addEventListener()` method.

```
const ele = document.getElementById('app');
ele.addEventListener('click', () => console.log('clicked'))
```

In Vue, we can use the `v-on:click` directive to implement a click handler. We can specify this click handler on an up-vote icon of a submission. We'll set this click event handler to call an `upvote(submission.id)` method whenever the up-vote icon is clicked. We'll pass in the `submission.id` as an argument to be used within the method. This updates the `div` element that encompasses the up-vote icon to this:

```
<div class="media-right">
  <span class="icon is-small" v-on:click="upvote(submission.id)">
    <i class="fa fa-chevron-up"></i>
    <strong class="has-text-info">{{ submission.votes }}</strong>
  </span>
</div>
```

Since we've specifed the click event, we now need to define the `upvote(submissionId)` method in our Vue instance.

A `methods` property exists in a Vue instance to allow us to define methods bound to that instance. Methods behave like normal JavaScript functions and are only evaluated when explicitly called. Below the `computed` property in our instance, let's introduce the `methods` property and the `upvote` method:

```
new Vue({
  el: '#app',
  data: {
    submissions: Seed.submissions
  },
  computed: {
    // ...,
  },
  methods: {
    upvote(submissionId) {
      const submission = this.submissions.find(
        submission => submission.id === submissionId
      );
      submission.votes++;
    }
```

```
  }
});
```

The up-voting logic involves using the native JavaScript `find()` method to locate the submission object with the `id` equal to the `submissionId` parameter. The `votes` attribute of that submission is then incremented by one.

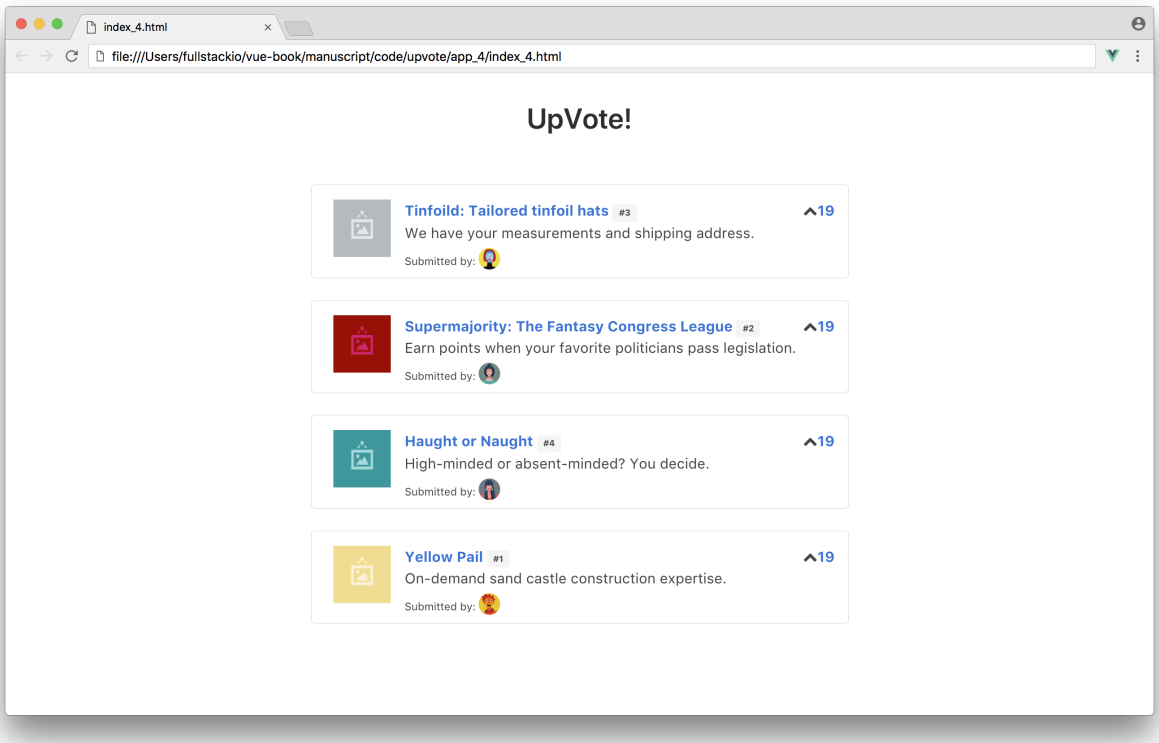
Reactive state

We need to note an **important** aspect of Vue here. With a library like React, the above method implementation is problematic since state is often treated as *immutable*. State within Vue, on the other hand, is *reactive*.

Reactive state is one of the key differences that makes Vue unique. State (i.e. data) management is often intuitive and easy to understand since modifying state often directly causes the view to update.

We'll be seeing more and more on how Vue data responds reactively throughout the book. For now, keep in mind Vue has an unobtrusive system to how data is modified and the view reacts.

Our app is now responsive to user interaction. Let's save the `index.html` and `main.js` files, refresh the browser, and start clicking the up-vote icons.



They work! Try up-voting a submission multiple times. Do you notice how it immediately jumps over other submissions with lower vote counts? This functionality works thanks to Vue's reactivity system.

As we up-vote a submission, we are directly modifying the `this.submissions` data array. Our computed property `sortedSubmissions` depends on `this.submissions`, so as the latter changes, so does the former.

Our view is reactive to `sortedSubmissions`. When changes happen to our computed property, our view re-renders to display that change!

Class bindings

Our application has implemented almost all the functionality we expected from the beginning.

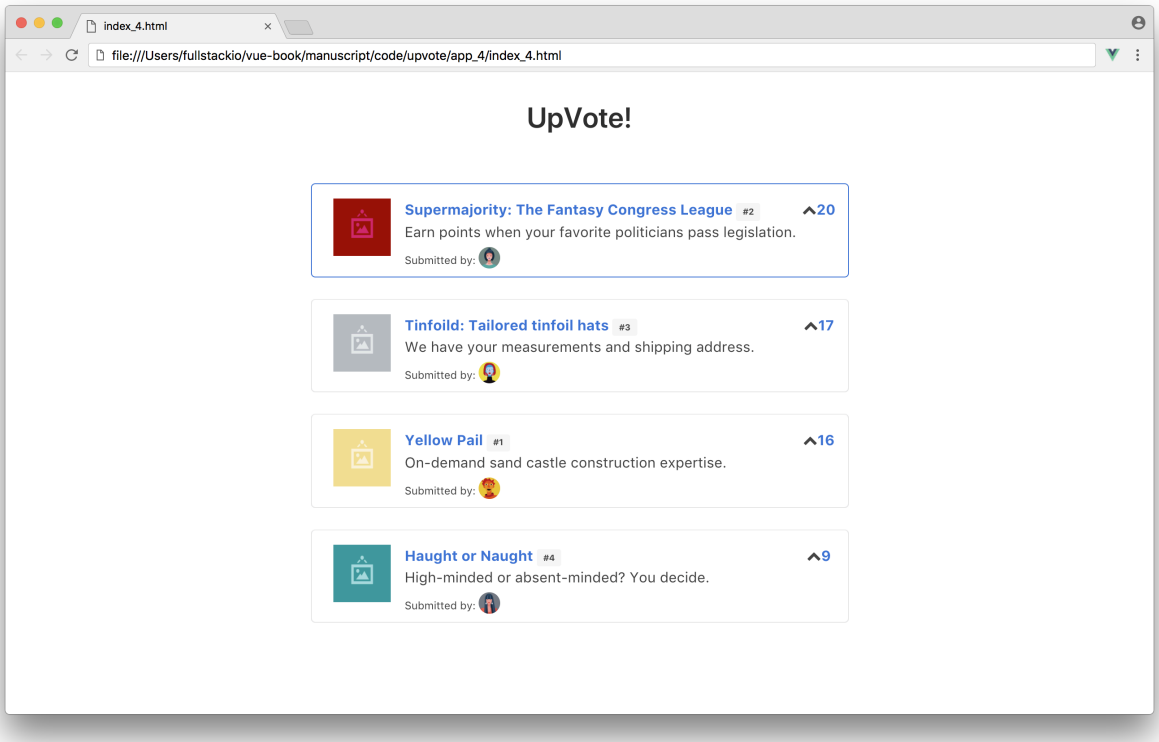
Before we dive in and try to improve how our code is laid out, let's add a conditional class that displays a special blue border around a submission when said submission reaches a certain number of votes (let's say 20 votes).

We have the class `blue-border` already set up in our custom `styles.css` file. Our conditional class binding will basically dictate: the presence of the `blue-border` class depends on the truthiness of `submission.votes >= 20`. We'll use the `v-bind` directive to dynamically enable the class when the submission votes exceeds 20:

```
<article v-for="submission in sortedSubmissions"
  v-bind:key="submission.id"
  class="media"
  v-bind:class="{ 'blue-border': submission.votes >= 20 }">
  <!-- Rest of the submission template -->
</article>
```

Pretty simple huh? There's many ways to specify inline conditional class and style bindings with which we'll investigate deeper throughout the rest of the book!

Now, if we go ahead and up-vote a submission to twenty or more votes, we'll see a blue border appear.



Yay! We've introduced all the features we initially had in mind for UpVote!. Our application is dynamically data-driven with external data, sorts all the submissions based on the number of votes, and listens for user interaction on up-voting.

Let's assume we had much larger plans on scaling the front end of UpVote!. New features could be added in like having a navigation header, a sidebar for submitting new submissions, a footer, etc. If we continue building our application the same way we've been going about it, we'll be introducing a lot more data/methods/properties to our Vue instance.

This will bloat our DOM, eventually making changes to our code unmanageable. This is where the concept of **isolated components** come in.

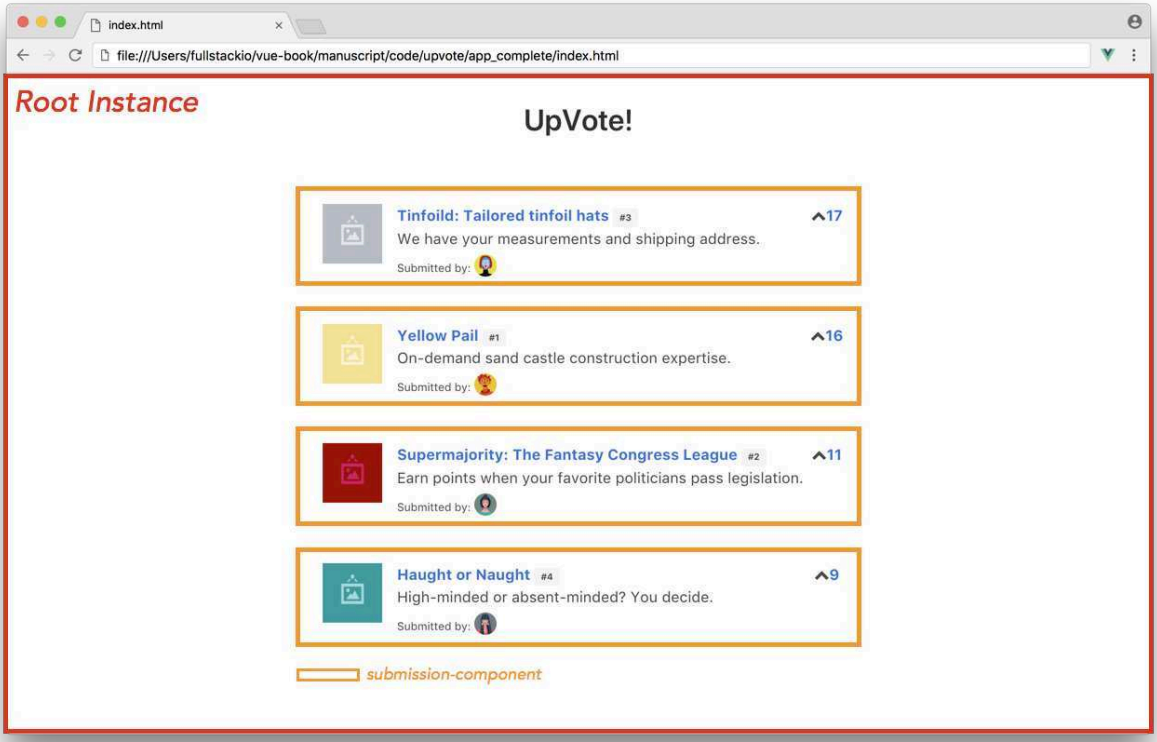
Components

Vue, like other modern-day JavaScript frameworks, provides the ability for users to create isolated components within their applications. **Reusability** and **maintainability** are some of the main reasons as to why components are especially important.

Components are intended to be **self-contained** modules since we can group markup (HTML), logic (JS) and even styles (CSS) within them. This allows for easier maintenance, especially when applications grow much larger in scale.

Let's create a new component for our application. As a result, we'll break apart the interface of our app into two separate entities:

- The parent component which encompasses all the separate submissions - this will be the existing Vue instance.
- The new submission component which represents a single submission.



Our app's components

The standard method for creating a global Vue component is handled by using the `Vue.component` constructor method:

```
Vue.component('submission-component', {
    // options
});
```

Though this would work, we'd want our component properly defined within the scope of our application instance. Instead, let's assign our newly created `submission-component` to a constant variable and register it as part of the `component` option in our Vue instance.

In our `main.js` file, let's specify a `submissionComponent` object that references a new component. We'll declare this object right above the root Vue instance:

```
const submissionComponent = {

};

new Vue({
  // ...
});
```

template

Vue components *are* Vue instances. The majority of properties (except for a few root-specific options) that exist in a root Vue instance (`data`, `methods`, etc.) can exist in a component as well.

In Vue instances, a `template` option exists that allows us to define the template of that instance. The simplest way of defining a template is within strings. Here's an example:

```
const submissionComponent = {
  template: '<div>Hello World!</div>'
}
```

If we wanted to define a template over multiple lines, we'll have to use ES6's template literals[30] (specified with the use of backticks). This is because JavaScript doesn't allow strings to span over multiple lines.

```
const submissionComponent = {
  template:
  ` <div>
      Hello World!
    </div>`
}
```

We're specifying templates for a Vue application that *isn't* being precompiled. In the next chapter[31], we'll be exposed to a different way of defining component templates since that chapter's application will be precompiled during build.

In the `submissionComponent`, the `template` property will reflect all the items contained within a single submission. Let's update the `submissionComponent` to reflect this:

[30] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals
[31] components

```
const submissionComponent = {
  template:
  `<div style="display: flex; width: 100%">
    <figure class="media-left">
      <img class="image is-64x64"
        v-bind:src="submission.submissionImage">
    </figure>
    <div class="media-content">
      <div class="content">
        <p>
          <strong>
            <a v-bind:href="submission.url" class="has-text-info">
              {{ submission.title }}
            </a>
            <span class="tag is-small">#{{ submission.id }}</span>
          </strong>
          <br>
            {{ submission.description }}
          <br>
          <small class="is-size-7">
            Submitted by:
            <img class="image is-24x24"
              v-bind:src="submission.avatar">
          </small>
        </p>
      </div>
    </div>
    <div class="media-right">
      <span class="icon is-small" v-on:click="upvote(submission.id)">
        <i class="fa fa-chevron-up"></i>
        <strong class="has-text-info">{{ submission.votes }}</strong>
      </span>
    </div>
  </div>`
};
```

There's a few things to address here.

1. The template of a component **must** be enclosed within a single root element. This is a **strict** limitation to declaring Vue templates. Because of this, we've wrapped everything within a `<div style="display: flex; width: 100%"></div>` element. We've added some additional styling to comply with this new root element.

2. The `submission` object in this template is currently `undefined`. When this component is declared, we're going have to pass data from the parent component (i.e. the root instance) down to this child component. We're going to use Vue **`props`** to pass data from the root instance to this component.
3. The `upvote()` click listener method needs to be mapped to a method within the `submissionComponent` for it to work. As a result, we're going to have to transfer the `upvote()` method from the Vue instance to this component.

Before we look into points (2) and (3), let's reference the newly created component in the DOM. In the `index.html` file; we'll first remove the submission template code within the `<article>` element. We'll then replace this inner content with a single `<submission-component>` element:

```
<article v-for="submission in sortedSubmissions"
  v-bind:key="submission.id"
  class="media"
  v-bind:class="{ 'blue-border': submission.votes >= 20 }">
  <submission-component></submission-component>
</article>
```

Our Vue instance currently doesn't recognize this `<submission-component>` element. In order to give the Vue instance awareness of our new component, we'll define it as a key in a `components` property of our Vue instance in the `main.js` file:

```
new Vue({
  // ...,
  components: {
    'submission-component': submissionComponent
  }
});
```

In the `components` options of the root Vue instance, we've mapped a `submission-component` declaration to the `submissionComponent` object.

Props

Vue gives us the ability to pass data from a parent component down to a child component with the help of **props**. In Vue, **props** are attributes that need to be given a value in the parent component and have to be explicitly declared in the child component. As a result, `props` can only flow in a single direction (parent to child), and never in the opposite direction (child to parent).

The `v-bind` directive is used to bind dynamic values (or objects) as props in a parent instance.

In the `index.html` file, we're going to pass both the iterated `submission` object and the `sortedSubmissions` array as `props` to `submission-component`. The `submission` object will be used in the template of

the `submission-component` while `sortedSubmissions` will be used in the `upvote()` method of that component.

This makes our `<article>` element be updated to this:

upvote/app_5/index.html

```
    <div class="section">
      <article v-for="submission in sortedSubmissions"
        v-bind:key="submission.id"
        class="media"
        v-bind:class="{ 'blue-border': submission.votes >= 20 }">
        <submission-component
          v-bind:submission="submission"
          v-bind:submissions="sortedSubmissions">
        </submission-component>
      </article>
    </div>
```

We've set the `submission` object to a prop of the same name and we've set the `sortedSubmissions` array to a prop labelled as `submissions`.

For a child component to use the props provided to it, it needs to explictly declare the props it receives with the `props` option. Let's introduce a `props` option in the `submissionComponent` object and specify the `submission` and `submissions` props being passed in:

```
const submissionComponent = {
  template:
  `
    // ...
  `,
  props: ['submission', 'submissions']
};
```

Now the `submission` object and the `submissions` array can safely be used within the template of `submissionComponent`. All that's left for us to do is migrate the `upvote()` component from the Vue instance to the `submissionComponent` object.

This will update the `submissionComponent` object to:

upvote/app_5/main.js

```
const submissionComponent = {
  template:
  ` <div style="display: flex; width: 100%">
      <figure class="media-left">
        <img class="image is-64x64"
          v-bind:src="submission.submissionImage">
      </figure>
      <div class="media-content">
        <div class="content">
          <p>
            <strong>
              <a v-bind:href="submission.url" class="has-text-info">
                {{ submission.title }}
              </a>
              <span class="tag is-small">#{{ submission.id }}</span>
            </strong>
            <br>
              {{ submission.description }}
            <br>
            <small class="is-size-7">
              Submitted by:
              <img class="image is-24x24"
                v-bind:src="submission.avatar">
            </small>
          </p>
        </div>
      </div>
      <div class="media-right">
        <span class="icon is-small" v-on:click="upvote(submission.id)">
          <i class="fa fa-chevron-up"></i>
          <strong class="has-text-info">{{ submission.votes }}</strong>
        </span>
      </div>
    </div>`,
  props: ['submission', 'submissions'],
  methods: {
    upvote(submissionId) {
      const submission = this.submissions.find(
        submission => submission.id === submissionId
      );
      submission.votes++;
    }
```

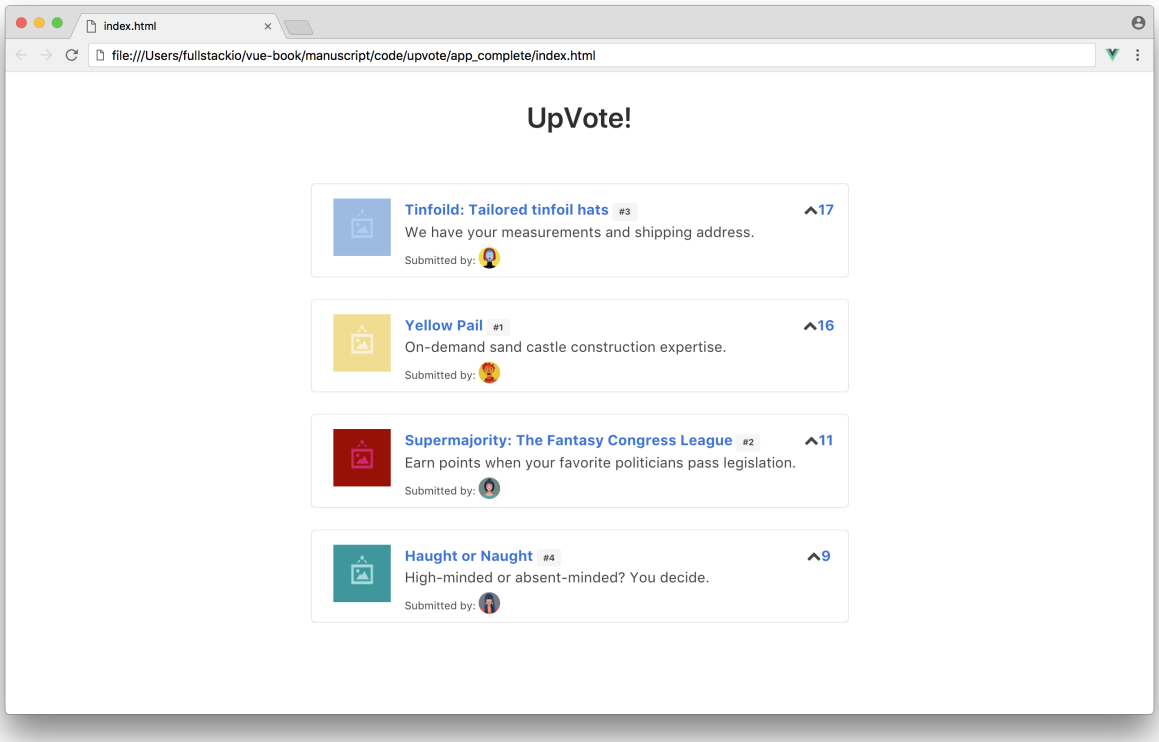
```
  }
};
```

And the Vue instance will now look like the following:

upvote/app_5/main.js

```
new Vue({
  el: '#app',
  data: {
    submissions: Seed.submissions
  },
  computed: {
    sortedSubmissions () {
      return this.submissions.sort((a, b) => {
        return b.votes - a.votes
      });
    }
  },
  components: {
    'submission-component': submissionComponent
  }
});
```

If we save the `main.js` file and refresh our browser, everything should remain as is and all functionality should work as expected!



`v-bind` and `v-on` shorthand syntax

Before we conclude this chapter, let's discuss another feature that Vue provides.

The `v-` prefix in Vue directives is a visual indicator that a Vue template attribute is being used. For simplicity, Vue provides shorthands for the commonly used `v-bind` and `v-on` directives.

The `v-bind` directive can be shortened with the `:` symbol:

```
// the full syntax
<img v-bind:src="submission.submissionImage" />

// the shorthand syntax
<img :src="submission.submissionImage" />
```

And the `v-on` directive can be shortened with the `@` symbol:

```
// the full syntax
<span v-on:click="upvote(submission.id)"></span>

// the shorthand syntax
<span @click="upvote(submission.id)"></span>
```

This shorthand syntax is completely optional but allows us to use the `v-bind` and `v-on` directives without explicitly typing out the full syntax.

> For the rest of the book we'll stick to using the shorthand syntax for `v-bind` and the `v-on` directives.

For the UpVote! application, you'll be able to see the use of the shorthand syntax in the `upvote/app_complete/` folder. **The rest of the code remains the same with the only changes replacing the `v-bind` and `v-on` syntax with `:` and `@` respectively**.

Congratulations!

We just wrote our first Vue app. We've gone through the easiest foray to getting started and there are plenty of powerful features we haven't covered yet. With this chapter, we've managed to understand the core fundamentals that we'll be building on throughout the book.

Recap

1. The Vue instance is the starting point of all Vue applications. The instance can have options like the `data`, `computed` and `methods` properties and is often mounted/attached to a DOM element.
2. The 'Mustache' syntax can be used for data binding. The `v-bind` directive is used for binding HTML attributes.
3. Vue directives such as `v-for` can be used to manipulate the template based on the data provided. The `v-on` directive is used as an event handler to listen to DOM events.
4. We think and organize our Vue apps with components.

Onward!

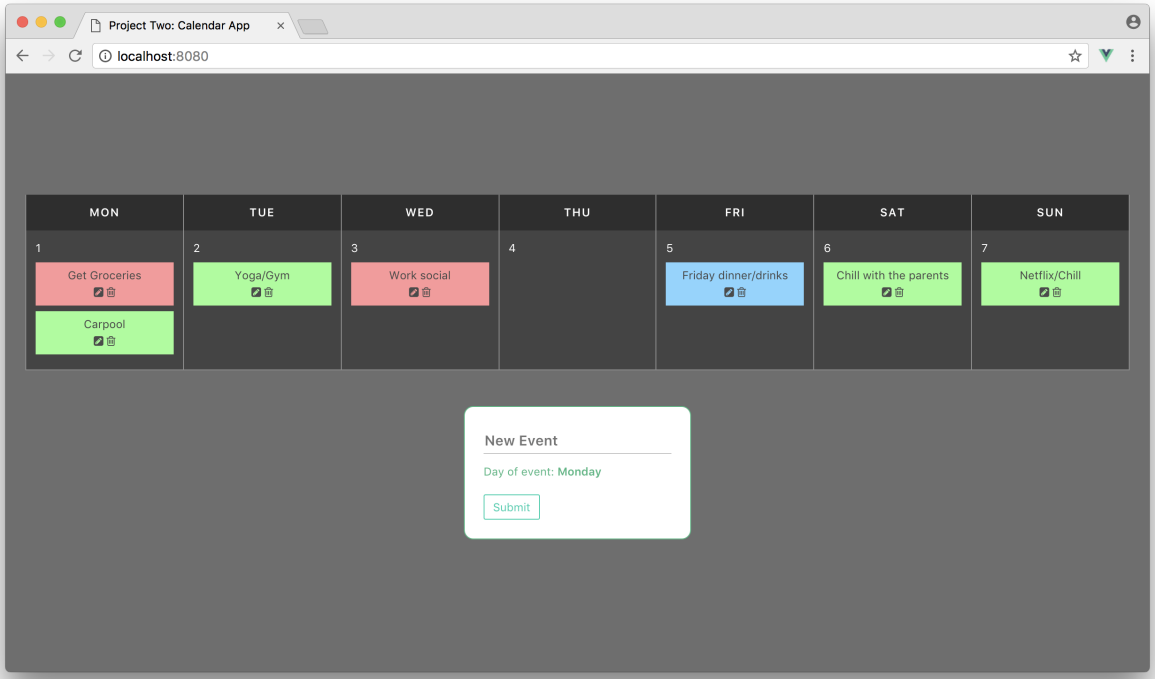
II - Single-file components

Introduction

In the last chapter we briefly covered how Vue lets us organize our app into components which can be used and manipulated in the view.

In this chapter, we'll be diving in deeper into building components with Vue. We'll investigate a pattern that we'll be be able to use to scale Vue apps from scratch. We'll be using this pattern to create an app interface that manages events within a weekly calendar.

In our weekly calendar app, a user can add, delete, and edit day to day events within a week. Each event corresponds to a particular task/to-do item that the user would like to keep track of:



Setting up our development environment

Node.js and npm

For this project (and the majority of projects) in this book, we'll need to make sure we have a working Node.js[32] development environment along with the Node Package Manager (`npm`).

There are a couple different ways we can install Node.js so please refer to the Node.js website for detailed information: https://nodejs.org/download/[33].

It's also possible to install Node.js using a tool like nvm[34] or the n[35] tool. Using a package like these allow us to maintain multiple version of node in our development environment.

If you're on a Mac, your best bet is to install Node.js directly from the Node.js[36] website instead of through another package manager (like Homebrew). Installing Node.js via Homebrew is known to cause some issues.

If you're on a Windows machine, you would need to install Node.js through the Windows Installer from the Node.js[37] website.

The easiest way to verify if Node.js has been successfully installed is to check which version of Node.js is running. To do this, we'll open a terminal window and run the following command:

```
$ node -v
```

npm is installed as a part of Node.js. To check if npm is available within our development environment, we can list the version of our npm binary with:

```
$ npm -v
```

In either case, if a version number is not printed out and instead an error is emitted, make sure to download a Node.js installer that includes npm and ensure that the `PATH` is set appropriately.

Vue syntax highlighting

In this chapter, we'll be introducing Vue single-file components. These components allow us to write Vue code within a new file format - `.vue`. Depending on your code editor, you may need to install a syntax highlighting plugin to simplify the readability of these components. Here are some popular Vue code highlighting plugins for the following editors:

[32] http://nodejs.org
[33] https://nodejs.org/download/
[34] https://github.com/creationix/nvm
[35] https://github.com/tj/n
[36] http://nodejs.org
[37] http://nodejs.org

GET THE FULL BOOK

This is the end of the preview chapter!

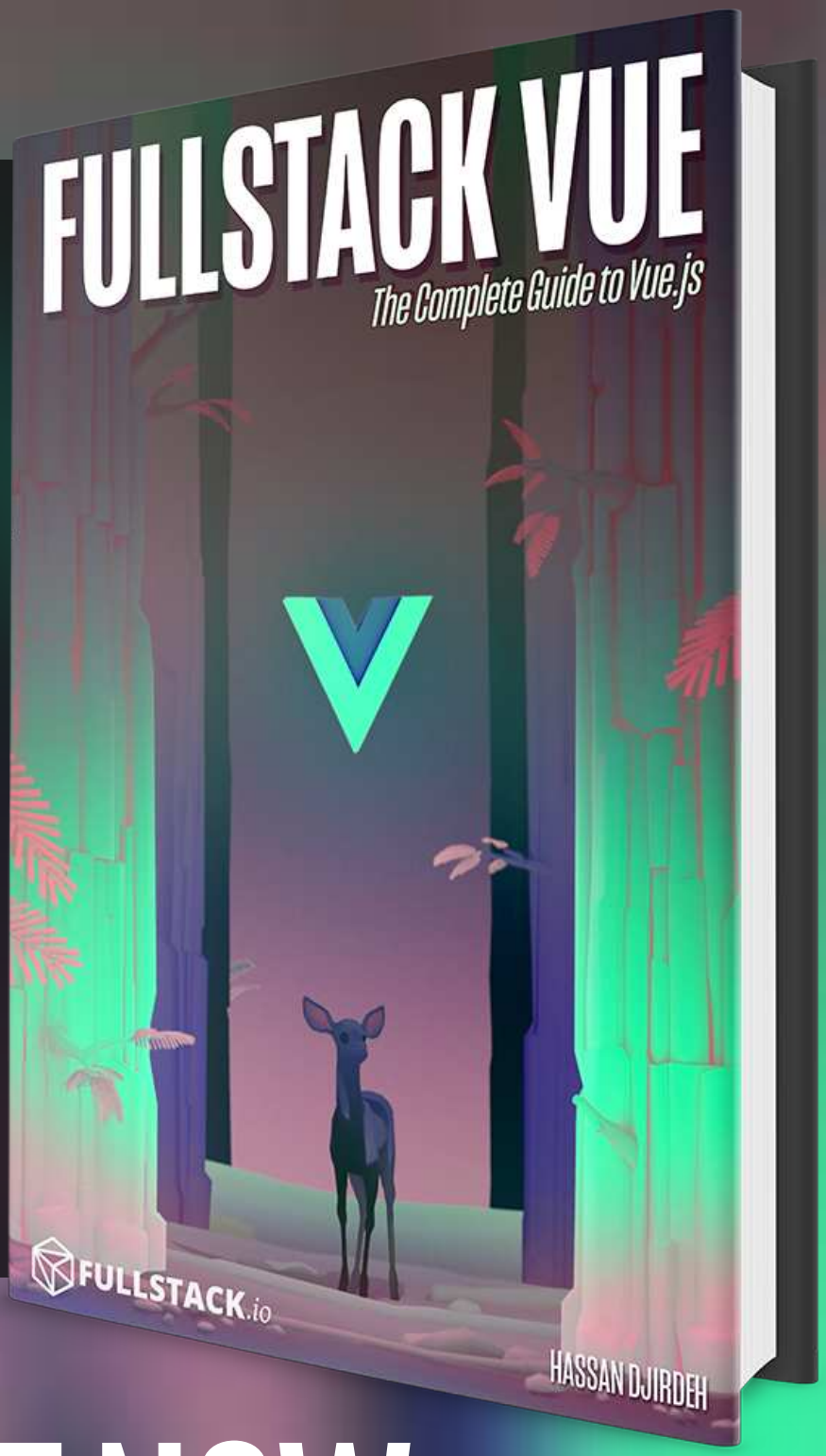
Head over to:

https://fullstack.io/vue

to download the full package!

Learn how to use:

- Events
- Routing
- Forms
- Vuex
- Testing
- and more!



GET IT NOW