The Majesty of Vue.js

Alex Kyriakidis

Kostas Maniatis

Contents

Building Large-Scale Applications 142

Introduction

About Vue.js

Vue.js Overview

Vue.js (pronounced /vju:/, like view) is a library for building interactive web interfaces. The goal of Vue.js is to provide the benefits of **reactive data binding** and **composable view components** with an API that is as simple as possible.

Vue.js itself is not a full-blown framework - it is focused on the view layer only. It is therefore very easy to pick up and to integrate with other libraries or existing projects. On the other hand, when used in combination with proper tooling and supporting libraries, Vue.js is also perfectly capable of powering sophisticated Single-Page Applications.

If you are an experienced frontend developer and you want to know how Vue.js compares to other libraries/frameworks, check out the Comparison with Other Frameworks chapter.

If you are interested to learn more information about Vue.js' core take a look at Vue.js official guide[1].

What people say about Vue.js

"Vue.js is what made me love JavaScript. It's extremely easy and enjoyable to use. It has a great ecosystem of plugins and tools that extend its basic services. You can quickly include it in any project, small or big, write a few lines of code and you are set. Vue.js is fast, lightweight and is the future of Front end development!"

—Alex Kyriakidis

"When I started picking up Javascript I got excited learning a ton of possibilities, but when my friend suggested to learn Vue.js and I followed his advice, things went wild. While reading and watching tutorials I kept thinking all the stuff I've done so far and how much easier it would be if I had invest time to learn Vue earlier. My opinion is that if you want to do your work fast, nice and easy Vue is the JS Framework you need. "

—Kostas Maniatis

---

[1] http://vuejs.org/guide/overview.html

"Mark my words: Vue.js will sky-rocket in popularity in 2016. It's that good."

— Jeffrey Way

"Vue is what I always wanted in a JavaScript framework. It's a framework that scales with you as a developer. You can sprinkle it onto one page, or build an advanced single page application with Vuex and Vue Router. It's truly the most polished JavaScript framework I've ever seen."

— Taylor Otwell

"Vue.js is the first framework I've found that feels just as natural to use in a server-rendered app as it does in a full-blown SPA. Whether I just need a small widget on a single page or I'm building a complex Javascript client, it never feels like not enough or like overkill."

— Adam Wathan

"Vue.js has been able to make a framework that is both simple to use and easy to understand. It's a breath of fresh air in a world where others are fighting to see who can make the most complex framework."

— Eric Barnes

"The reason I like Vue.js is because I'm a hybrid designer/developer. I've looked at React, Angular and a few others but the learning curve and terminology has always put me off. Vue.js is the first JS framework I understand. Also, not only is it easy to pick up for the less confidence JS'ers, such as myself, but I've noticed experienced Angular and React developers take note, and liking, Vue.js. This is pretty unprecedented in JS world and it's that reason I started London Vue.js Meetup."

—Jack Barham

Comparison with Other Frameworks

Angular

There are a few reasons to use Vue over Angular, although they might not apply for everyone:

- Vue.js is much simpler than Angular, both in terms of API and design. You can learn almost everything about it really fast and get productive.
- Vue.js is a more flexible, less opinionated solution. That allows you to structure your app the way you want it to be, instead of being forced to do everything the in Angular way. It's only an interface layer so you can use it as a light feature in pages instead of a full blown SPA. It gives you a bigger room to mix and match with other libraries, but you are also responsible for making more architectural decisions. For example, Vue.js' core doesn't come with routing or ajax functionalities by default, and usually assumes you are building the application using an external module bundler. This is probably the most important distinction.
- Angular uses two-way binding between scopes. While Vue also supports explicit two-way bindings, it defaults to a one-way, parent-to-child data flow between components. Using one-way binding makes the flow of data easier to reason about in large apps.
- Vue.js has a clearer separation between directives and components. Directives are meant to encapsulate DOM manipulations only, while Components stand for a self-contained unit that has its own view and data logic. In Angular there's a lot of confusion between the two.
- Vue.js has better performance and is much, much easier to optimize, because it doesn't use dirty checking. Angular gets slow when there are a lot of watchers, because every time anything in the scope changes, all these watchers need to be re-evaluated again. Also, the digest cycle may have to run multiple times to "stabilize" if some watcher triggers another update. Angular users often have to resort to esoteric techniques to get around the digest cycle, and in some situations there's simply no way to optimize a scope with a large amount of watchers. Vue.js doesn't suffer from this at all because it uses a transparent dependency-tracking observing system with async queueing - all changes trigger independently unless they have explicit dependency relationships. The only optimization hint you'll ever need is the `track-by` param on `v-for` lists.

Interestingly, there are quite some similarities in how Angular 2 and Vue are addressing these Angular 1 issues.

React

React and Vue.js do share a similarity in that they both provide reactive & composable View components. There are, of course, many differences as well.

First, the internal implementation is fundamentally different. React's rendering, leverages the Virtual DOM, an in-memory representation of what the actual DOM should look like. When the state changes, React does a full re-render of the Virtual DOM, diffs it, and then patches the real DOM.

The virtual-DOM approach provides a functional way to describe your view at any point of time, which is really nice. The view is by definition guaranteed to be in sync with the data, because it doesn't use observables, and re-renders the entire app on every update. . It also opens up possibilities to isomorphic JavaScript applications.

Instead of a Virtual DOM, Vue.js uses the actual DOM as the template and keeps references to actual nodes for data bindings. This, limits Vue.js to environments where DOM is present. However, contrary to the common misconception that Virtual-DOM makes React faster than anything else, Vue.js actually out-performs React when it comes to hot updates, and requires almost no hand-tuned optimization. With React, you need to implement `shouldComponentUpdate` everywhere, or use immutable data structures to achieve fully optimized re-renders.

API-wise, one issue with React (or JSX) is that the render function often involves a lot of logic, and ends up looking more like a piece of program (which in fact it is) rather than a visual representation of the interface. For some developers this is a bonus, but for designer/developer hybrids like me, having a template makes it much easier to think visually about the design and CSS. JSX mixed with JavaScript logic, breaks that visual model I need to map the code to the design. In contrast, Vue.js pays the cost of a lightweight data-binding DSL so that we have a visually scannable template and with logic encapsulated into directives and filters.

Another issue with React is that because DOM's updates are completely delegated to the Virtual DOM, it's a bit tricky when you actually **want** to control the DOM yourself (although theoretically you can, you'd be essentially working against the library when you do that). For applications that need ad-hoc custom DOM manipulations, especially animations with complex timing requirements, this can become a pretty annoying restriction. On this front, Vue.js allows for more flexibility and there are multiple FWA/Awwwards winning sites[2] built with Vue.js.

Some additional notes:

- The React team has very ambitious goals in making React a platform-agnostic UI development paradigm, while Vue is focused on providing a pragmatic solution for the web.
- React, due to its functional nature, plays very well with functional programming patterns. However it also introduces a higher learning barrier for junior developers and beginners. In this regard, Vue is much easier to pick up and get productive with.

[2] https://github.com/vuejs/vue/wiki/Projects-Using-Vue.js#interactive-experiences

- For large applications, the React community has been doing a lot of innovation in terms of state management solutions, e.g. Flux/Redux. Vue itself doesn't really address that problem (same for React core), but the state management patterns can be easily adopted for a similar architecture. Vue has its own state management solution called Vuex[3], and it's also possible to use Redux with Vue[4].
- The trend in React development is pushing you to put everything in JavaScript, including your CSS. There have been many CSS-in-JS solutions out there, but all -more or less- have their own problems. Most important, it deviates from the standard CSS authoring experience and makes it very awkward to leverage existing work in the CSS community. Vue's single file components[5] gives you component-encapsulated CSS while still allowing you to use your pre-processors of choice.

[3] https://github.com/vuejs/vuex
[4] https://github.com/egoist/revue
[5] http://vuejs.org/guide/application.html#Single_File_Components

Ember

Ember is a full-featured framework that is designed to be highly opinionated. It provides a lot of established conventions, and once you are familiar enough with them, it can make you very productive. However, it also means that the learning curve is high and that the flexibility suffers. It's a trade-off when you try to pick between an opinionated framework and a library with a loosely coupled set of tools that work together. The latter gives you more freedom but also requires you to make more architectural decisions.

That said, it would probably make a good comparison between Vue.js' core and Ember's templating and object model layer:

- Vue provides unobtrusive reactivity on plain JavaScript objects, and fully automatic computed properties. In Ember you need to wrap everything in Ember Objects and manually declare dependencies for computed properties.
- Vue's template syntax harnesses the full power of JavaScript expressions, while Handlebars' expression and helper syntax is quite limited in comparison.
- Performance wise, Vue outperforms Ember by a fair margin. Even after the latest Glimmer engine update in Ember 2.0, Vue automatically batches updates. On the other hand, in Ember, you need to manually manage run loops in performance-critical situations.

Polymer

Polymer is yet another Google-sponsored project and in fact was a source of inspiration for Vue.js as well. Vue.js' components can be loosely compared to Polymer's custom elements, and both provide a very similar development style. The biggest difference is that Polymer is built upon the latest Web Components features, and requires non-trivial polyfills to work (with degraded performance) in browsers that don't support those features natively. In contrast, Vue.js works without any dependencies down to IE9.

Also, in Polymer 1.0 the team has really made its data-binding system very limited in order to compensate for the performance. For example, the only expressions supported in Polymer templates, are the boolean negation and single method calls. Its computed property implementation is also not very flexible.

Finally, when deploying to production, Polymer elements need to be bundled via a Polymer-specific tool called vulcanizer. In comparison, single file Vue components can leverage everything the Webpack ecosystem has to offer. Thus, in your Vue components you can easily use ES6 and any of the CSS pre-pocessors you'd like.

Riot

Riot 2.0 provides a similar component-based development model (which is called a "tag" in Riot), with a minimal and beautifully designed API. I think Riot and Vue share a lot in design philosophies. However, despite being a bit heavier than Riot, Vue does offer some significant advantages over Riot:

- True conditional rendering (Riot renders all if branches and simply show/hide them)
- A far-more powerful router (Riot's routing API is just way too minimal)
- More mature tooling support (see webpack + vue-loader)
- Transition effect system (Riot has none)
- Better performance. (Riot in fact uses dirty checking rather than a virtual-dom, and thus suffers from the same performance issues with Angular.)

For updated comparisons feel free to check Vue.js guide.

Welcome

About the Book

This book will guide you through the path of the rapidly spreading Javascript Framework called Vue.js!

Some time ago, we started a new project based on Laravel and Vue.js. After thoroughly reading Vue.js guide and a few tutorials, we discovered lack of resources about Vue.js around the web. During the development of our project, we gained a lot of experience, so we came up with the idea to write this book in order to share our acquired knowledge with the world.

This book is written in an informal, intuitive, and easy-to-follow format, wherein all examples are appropriately detailed enough to provide adequate guidance to everyone.

We'll start from the very basics and through many examples we'll cover the most significant features of Vue.js. By the end of this book you will be able to create fast front end applications and increase the performance of your existing projects with Vue.js integration.

Who is this Book for

Everyone who has spent time to learn modern web development has seen Bootstrap, Javascript, and many Javascript frameworks. This book is for anyone interested in learning a lightweight and simple Javascript framework. No excessive knowledge is required, though it would be good to be familiar with HTML and Javascript. If you dont't know what the difference is between a string and an object, maybe you need to do some digging first.

This book is also useful for any reader who already know their way around Vue.js and want to expand their knowledge.

Get In Touch

In case you would like to contact us about the book, send us feedback, or other matters you would like to bring to our attention, don't hesitate to contact us.

Name	Email	Twitter
The Majesty of Vue.js	hello@tmvuejs.com	@tmvuejs
Alex Kyriakidis	alex@tmvuejs.com	@hootlex
Kostas Maniatis	kostas@tmvuejs.com	@kostaskafcas

Homework

The best way to learn code is to write code, so we have prepared one exercise at the end of most chapters for you to solve and actually test yourself on what you have learned. We strongly recommend you to try as much as possible to solve them and through them gain a better understanding of Vue.js. Don't be afraid to test your ideas, a little effort goes a long way! Maybe a few different examples or ways will give you the right idea. Of course we are not merciless, hints and potential solutions will be provided!

You may begin your journey!

Sample Code

You can find most of the code examples used in the book on GitHub. You can browse around the code here[6].

If you prefer to download it, you will find a `.zip` file here[7].

This will save you from copying and pasting things out of the book, which would probably be terrible.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in the book we would be grateful if you could report it to us. By doing so, you can protect other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please submit an issue on our GitHub repository[8].

Conventions

The following notational conventions are used throughout the book.

A block of code is set as follows:

JavaScript

[6] https://github.com/hootlex/the-majesty-of-vuejs

[7] https://github.com/hootlex/the-majesty-of-vuejs/archive/v1.0.1.zip

[8] https://github.com/hootlex/the-majesty-of-vuejs

```
function(x, y){
    // this is a comment
}
```

Code words in text, data are shown as follows: "Use `.container` for a responsive fixed width container."

New terms and **important words** are shown in bold.

Tips, notes, and warnings are shown as follows:

This is a Warning

This element indicates a warning or caution.

This is a Tip

This element signifies a tip or suggestion.

This is an Information box

Some special information here.

This is a Note

A note about the subject.

This is a Hint

A hint about the subject.

This is a Terminal Command

Commands to run in terminal.

This is a Comparison text

A small text comparing things relative to the subject.

This is a link to Github for the code examples

Links which lead to the repository of this book, where you can find the code samples of each chapter.

Vue.js Fundamentals

1. Install Vue.js

When it comes to download Vue.js you have a few options to choose from.

1.1 Standalone Version

1.1.1 Download from vuejs.org

To install Vue you can simply download and include it with a script tag. `Vue` will be registered as a global variable.

You can download two versions of Vue.js:

1. Development Version from http://vuejs.org/js/vue.js[1]
2. Production Version from http://vuejs.org/js/vue.min.js[2].

Tip: Don't use the minified version during development. You will miss out all the nice warnings for common mistakes.

1.1.2 Include from CDN

You can find Vue.js also on jsdelivr[3] or cdnjs[4]

It takes some time to sync with the latest version so you have to check frequently for updates.

1.2 Download using NPM

NPM is the recommended installation method when building large scale apps with Vue.js. It pairs nicely with a CommonJS module bundler such as Webpack[5] or Browserify[6].

[1] http://vuejs.org/js/vue.js
[2] http://vuejs.org/js/vue.min.js
[3] http://cdn.jsdelivr.net/vue/1.0.26/vue.min.js
[4] https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.min.js
[5] http://webpack.github.io/
[6] http://browserify.org/

```
# latest stable
$ npm install vue
# latest stable + CSP-compliant
$ npm install vue@csp
# dev build (directly from GitHub):
$ npm install vuejs/vue#dev
```

1.3 Download using Bower

```
# latest stable
$ bower install vue
```

For more installation instructions and updates take a loot at the Vue.js Installation Guide[7]

In most book examples we are including Vue.js from the cdn, although you are free to install it using any method you like.

[7] http://vuejs.org/guide/installation.html

2. Getting Started

Let's start with a quick tour of Vue's data binding features. We're going to make a simple application that will allow us to enter a message and have it displayed on the page in real time. It's going to demonstrate the power of Vue's two-way data binding. In order to create our Vue application, we need to do a little bit of setting up, which just involves creating an HTML page.

In the process you will get the idea of the amount of time and effort we save using a javascript Framework like Vue.js instead of a javascript tool (library) like jQuery.

2.1 Hello World

We will create a new file and we will drop some boilerplate code in. You can name it anything you like, this one is called **`hello.html`**.

```
<html>
<head>
    <title>Hello Vue</title>
</head>
<body>
    <h1>Greetings your Majesty!</h1>
</body>
</html>
```

This is a simple HTML file with a greeting message.

Now we will carry on and do the same job using Vue.js. First of all we will include Vue.js and create a new Instance.

```
<html>
<head>
    <title>Hello Vue</title>
</head>
<body>
    <div id="app">
        <h1>Greetings your majesty!</h1>
    </div>
</body>
```

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.min.js">

</script>
<script>
    new Vue({
        el: '#app',
    })
</script>
</html>
```

For starters, we have included Vue.js from cdnjs[1] and inside a **script** tag we have our Vue instance. We use a **div** with an **id** of #app which is the element we refer to, so Vue knows where to 'look'. Try to think of this as a container that Vue works at. Vue won't recognize anything outside of the targeted element. Use the **el** option to target the element you want.

Now we will assign the message we want to display, to a variable inside an object named **data**. Then we'll pass the data object as an option to Vue constructor.

```
var data = {
    message: 'Greetings your majesty!'
};
new Vue({
    el: '#app',
    data: data
})
```

To display our message on the page, we just need to wrap the message in double curly brackets . So whatever is inside our message it will appear automatically in the **h1** tag.

```
<div id="app">
    <h1>{{ message }}</h1>
</div>
```

It is as simple as that. Another way to define the message variable is to do it directly inside Vue constructor in **data** object.

[1] https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.min.js

```
new Vue({
    el: '#app',
    data: {
        message: 'Greetings your Majesty!'
    }
});
```

Both ways have the exact same result, so you are again free to pick whatever syntax you like.

Info

The double curly brackets are not HTML but scripting code, anything inside mustache tags are called binding expressions. Javascript will evaluate these expressions. The **`{{ message }}`** brings up the value of the Javascript variable. This piece of code **`{{1+2}}`** will display the number **3**.

2.2 Two-way Binding

What is cool about Vue is that it makes our lives easier. Say we want to change the message on user input, how can we easily accomplish it? In the example below we use **`v-model`**, a directive of Vue (we will cover more on directives in the next chapter). Then we use two-way data binding to dynamically change the message value when the user changes the message text inside an input. Data is synced on every input event by default.

```
<div id="app">
    <h1>{{ message }}</h1>
    <input v-model="message">
</div>
```

```
new Vue({
    el: '#app',
    data: {
        message: 'Greetings your Majesty!'
    }
})
```

That's it. Now our heading message and user input are binded! By using **`v-model`** inside the **`input`** tag we tell Vue which variable should bind with that **`input`**, in this case **`message`** .

Two-way data binding

Two-way data binding means that if you change the value of a model in your view, everything will be kept up to date.

2.3 Comparison with jQuery.

Probably all of you have a basic experience with jQuery. If you don't, it's okay, the use of jQuery in this book is minimal. When we use it, its only to demonstrate how things can be done with Vue instead of jQuery and we will make sure everybody gets it.

Anyway, in order to better understand how data-binding is helping us to build apps, take a moment and think how you could do the previous example using jQuery. You would probably create an input element and give it an `id` or a `class` so you could target it and modify it accordingly. After this, you would call a function that changes the desired element to match the input value, whenever the *keyup event* happens. **It's a real bother**.

More or less, your snippet of code would look like this.

```
<html>
<head>
    <title>Hello Vue</title>
</head>
<body>
<div id="app">
    <h1>Greetings your Majesty!</h1>
    <input id="message">
</div>
</body>
<script src="https://code.jquery.com/jquery-2.1.4.min.js"></script>
<script type="text/javascript">
    $('#message').on('keyup', function(){
        var message = $('#message').val();
        $('h1').text(message);
    })
</script>
</html>
```

This is a simple example of comparison and, as you can see, Vue appears to be much more beautiful, less time consuming, and easier to grasp. Of course, jQuery is a powerful JavaScript library for Document Object Model (DOM) manipulation but everything comes with its ups and downs!

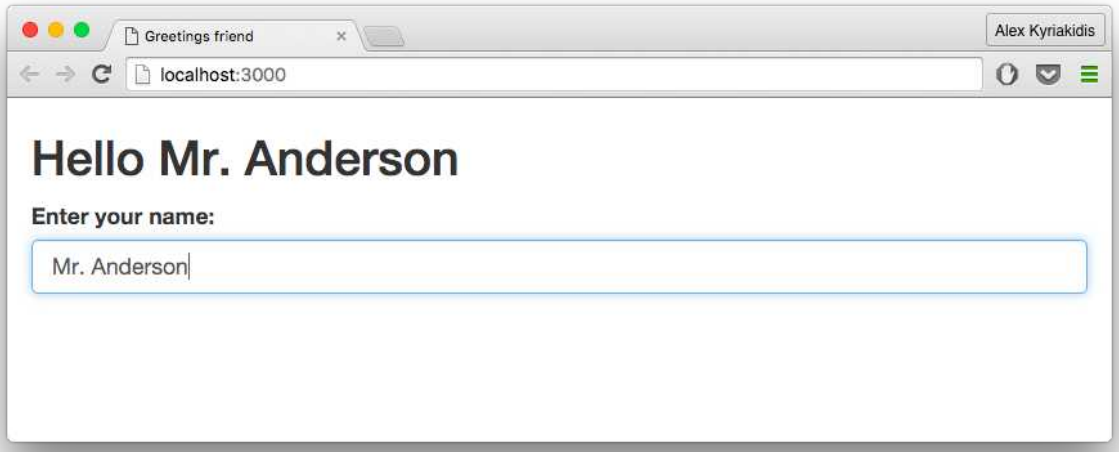
Code Examples

You can find the code examples of this chapter on GitHub[2].

[2] https://github.com/hootlex/the-majesty-of-vuejs/tree/master/examples/2.%20Getting%20started

2.4 Homework

A nice and super simple introductory exercise is to create an HTML file with a Hello, `{{name}}` heading. Add an input and bind it to `name` variable. As you can imagine, the heading must change instantly whenever the user types or changes his name. Good luck and have fun!



Example Output

Note

The example's output makes use of Bootstrap. If you are not familiar with bootstrap you can ignore it for now, it is covered in a later chapter.

You can find a potential solution to this exercise here[3].

[3] https://github.com/hootlex/the-majesty-of-vuejs/blob/master/homework/chapter2.html

3. A Flavor of Directives.

In this chapter we are going through some basic examples of Vue's directives. Well, if you have not used any Framework like Vue.js or AngularJS before, you probably don't know what a directive is. Essentially, a directive is some special token in the markup that tells the library to do something to a DOM element. In Vue.js, the concept of directive is drastically simpler than that in Angular. Some of the directives are:

- **`v-show`** which is used to conditionally display an element
- **`v-if`** which can be used instead of **`v-show`**
- **`v-else`** which displays an element when **`v-if`** or **`v-show`** evaluates to false.

Also, there is **`v-for`**, which requires a special syntax and its use is for rendering (e.g. render a list of items based on an array). We will elaborate about the use of each later in this book.

Let us begin and take a look at the directives we mentioned.

3.1 v-show

To demonstrate the first directive we are going to build something simple. We will give you some tips that will make your understanding and work much easier! Suppose you find yourself in need to toggle the display of an element, based upon some set of criteria. Maybe a submit button shouldn't display unless you've first typed in a message. How can we accomplish that with Vue?

```
<html>
<head>
    <title>Hello Vue</title>
</head>
<body>
<div id="app">
    <textarea></textarea>
</div>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
<script>
    new Vue({
        el: '#app',
        data: {
```

```
15              message: 'Our king is dead!'
16          }
17      })
18  </script>
19  </html>
```

Here we have an HTML file with our known **`div id="app"`** and a **`textarea`**. Inside the **`textarea`** we are going to display our message. Of course, it is not yet binded and by this point you may have already figured it out. Also you may have noticed that in this example we are no longer using the minified version of Vue.js. As we have mentioned before, the minified version shouldn't be used during development because you will miss out warnings for common mistakes. From now on we are going to use this version in the book but of course you are free to do as you like.

```
1   <html>
2   <head>
3       <title>Hello Vue</title>
4   </head>
5   <body>
6   <div id="app">
7       <textarea v-model="message"></textarea>
8       <pre>
9           {{$data | json}}
10      </pre>
11  </div>
12  </body>
13  <script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
14  <script>
15      new Vue({
16          el: '#app',
17          data: {
18              message: 'Our king is dead!'
19          }
20      })
21  </script>
22  </html>
```

It is time to bind the value of **`textarea`** with our **`message`** variable using **`v-model`** so it displays our message. Anything we type in is going to change in real time, just as we saw in the example from the previous chapter, where we were using an input. Additionally, here we are using a **`pre`** tag to spit out the data. What this is going to do, is to take the data from our Vue instance, filter it through **`json`**, and finally display the data in our browser. We believe, that this gives a much better way to

build and manipulate our data, since having everything right in front of you is better than looking constantly at your console.

Info

JSON (JavaScript Object Notation) is a lightweight data-interchange format. You can find more info on JSON here[1]. The output of **{{$data | json}}** is binded with Vue data and will get updated on every change.

```
<html>
<head>
    <title>Hello Vue</title>
</head>
<body>
<div id="app">
    <h1>You must send a message for help!</h1>
    <textarea v-model="message"></textarea>
    <button v-show="message">
        Send word to allies for help!
    </button>
    <pre>
        {{$data | json}}
    </pre>
</div>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
<script>
    new Vue({
        el: '#app',
        data: {
             message: 'Our king is dead! Send help!'
        }
    })
</script>
</html>
```

Carrying on, we now have a simple warning in the **h1** tag that will toggle later based on some criteria. Next to it, there is the button which is going to display conditionally. It appears only if there is a message present. If the **textarea** is empty and therefore our data, the button's **display** attribute is automatically set to 'none' and the button disappears.

[1] http://www.json.org/

Info

An element with **`v-show`** will always be rendered and remain in the DOM. **`v-show`** simply toggles the **`display`** CSS property of the element.

```
<h1 v-show="!message">You must send a message for help!</h1>
<textarea v-model="message"></textarea>
<button v-show="message">
    Send word to allies for help!
</button>
```

What we want to accomplish in this example, is to toggle different elements. In this step, we need to hide the warning inside the **`h1`** tag, **if** a message is present. Otherwise hide the message by setting its **`style`** to **`display: none`**.

3.2 v-if

In this point you might ask 'What about the v-if directive we mentioned earlier?'. So, at this point we will build the previous example again, only this time we'll use **`v-if`**!

```
<html>
<head>
    <title>Hello Vue</title>
</head>
<body>
<div id="app">
    <h1 v-if="!message">You must send a message for help!</h1>
    <textarea v-model="message"></textarea>
    <button v-show="message">
        Send word to allies for help!
    </button>
    <pre>
        {{$data | json}}
    </pre>
</div>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
<script>
    new Vue({
        el: '#app',
```

```
        data: {
             message: 'Our king is dead! Send help!'
        }
    })
</script>
</html>
```

As shown, the replacement of **v-show** with **v-if** works just as good as we thought. Go ahead and try to make your own experiments to see how this works! The only difference is that an element with **v-if** will not remain in the DOM.

3.2.1 Template v-if

If sometime we find ourselves in a position where we want to toggle the existence of multiple elements at once, then we can use **v-if** on a **<template>** element. In occasions where the use of **div** or **span** seems appropriate, the **<template>** element can serve also as an invisible wrapper. Also the **<template>** won ‘t be rendered in the final result.

```
<div id="app">
    <template v-if="!message">
      <h1>You must send a message for help!</h1>
      <p>Dispatch a messenger immediately!</p>
      <p>To nearby kingdom of Hearts!</p>
    </template>
    <textarea v-model="message"></textarea>
    <button v-show="message">
        Send word to allies for help!
    </button>
    <pre>
        {{$data | json}}
    </pre>
</div>
```

Template v-if

Using the setup from the previous example we have attached the **v-if** directive to the **template** element, toggling the existence of all nested elements.

Warning

The **v-show** directive does not support the **<template>** syntax.

3.3 v-else

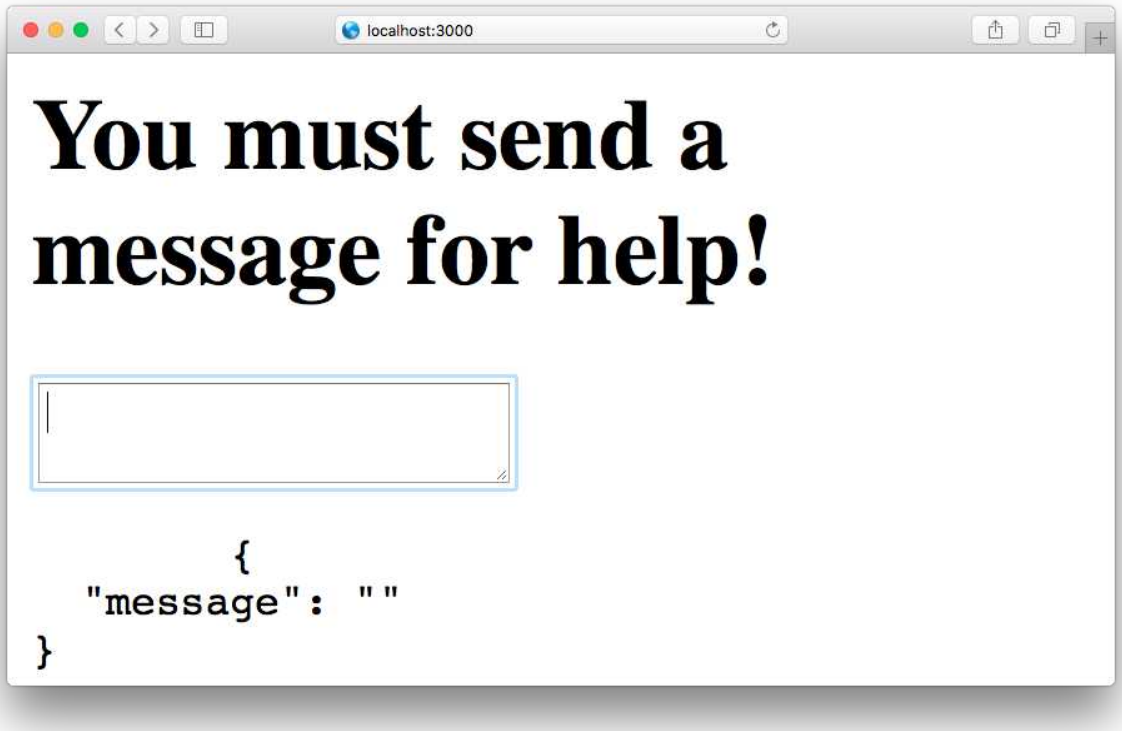
When using **v-if** or **v-show** you can use the **v-else** directive to indicate an "else block" as you might have already imagined. Be aware that the **v-else** directive must follow immediately the **v-if** or **v-show** directive - otherwise it will not be recognized.

Using **v-else** with **v-show**.

```
<html>
<head>
    <title>Hello Vue</title>
</head>
<body>
<div id="app">
    <h1 v-show="!message">You must send a message for help!</h1>
    <h2 v-else>You have sent a message!</h2>
    <textarea v-model="message"></textarea>
    <button v-show="message">
        Send word to allies for help!
    </button>
    <pre>
        {{$data | json}}
    </pre>
</div>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>

<script>
    new Vue({
        el: '#app',
        data: {
             message: 'Our king is dead! Send help!'
        }
    })
</script>
</html>
```

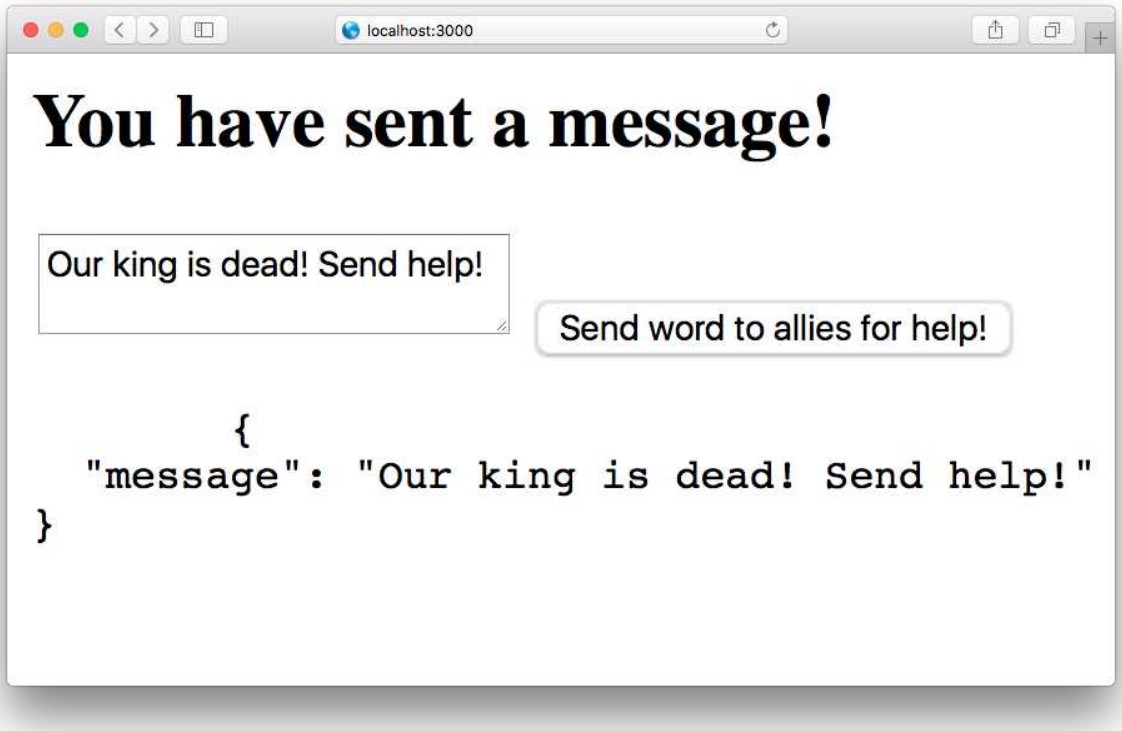
Using **v-else** with **v-if**.

```
<html>
<head>
    <title>Hello Vue</title>
</head>
<body>
<div id="app">
    <h1 v-if="!message">You must send a message for help!</h1>
    <h2 v-else>You have sent a message!</h2>
    <textarea v-model="message"></textarea>
    <button v-show="message">
        Send word to allies for help!
    </button>
    <pre>
        {{$data | json}}
    </pre>
</div>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>

<script>
    new Vue({
        el: '#app',
        data: {
             message: 'Our king is dead! Send help!'
        }
    })
</script>
</html>
```



v-if in action

v-if in action



v-else in action

v-else in action

v-else in action

Just for the sake of the example we have used an `h2` tag with a different warning than before, which is displayed conditionally. If no message is presented, we see the `h1` tag. If there is a message, we see the `h2` using this very simple syntax of Vue `v-if` and `v-else`. As you can see above we've used `v-if` as well as `v-show`. Both give us the same result. Simple as a pimple!

3.4 v-if vs. v-show

Even though we have already mentioned a difference between `v-if` and `v-show` , we can deepen a bit more. Some questions may arise out of their use. Is there a big difference between using `v-show` and `v-if`? Is there a situation where performance is affected? Are there problems where you're better off using one or the other? You might experience that the use of `v-show` on a lot of situations causes bigger time of load during page rendering. In comparison, `v-if` is truly conditional according to the guide of Vue.js.

> *When using `v-if`, if the condition is false on initial render, it will not do anything - partial compilation won't start until the condition becomes true for the first time. Generally speaking, `v-if` has higher toggle costs while `v-show` has higher initial render costs. So prefer `v-show` if you need to toggle something very often, and prefer `v-if` if the condition\ is unlikely to change at runtime.*

So, when to use which really depends on your needs.

Code Examples

You can find the code examples of this chapter on GitHub[2].

[2] https://github.com/hootlex/the-majesty-of-vuejs/tree/master/examples/3.%20A%20Flavor%20Of%20Directives

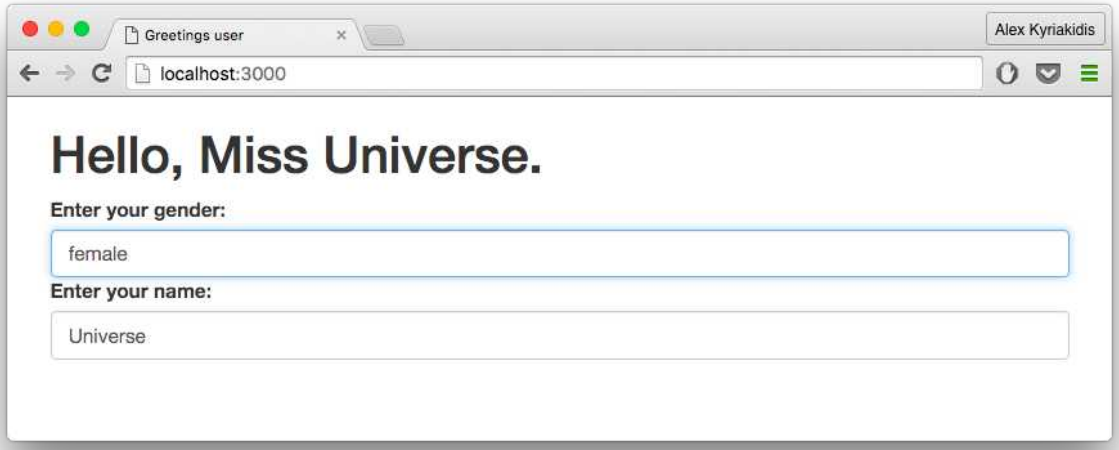
3.5 Homework

Following the previous homework exercise, you should try to expand it a bit. The user now types in his gender along with his name. If user is a male, then the heading will greet the user with "**Hello Mister {{name}}**". If user is a female, then "**Hello Miss {{name}}**" should appear instead.

When gender is neither male or female then the user should see the warning heading "**So you can't decide. Fine!**".

Hint

A logical operator would come handy to determine user title.



Example Output

Note

The example output makes use of Bootstrap. If you are not familiar with bootstrap you can ignore it for now, it is covered in a later chapter.

You can find a potential solution to this exercise here[3].

[3] https://github.com/hootlex/the-majesty-of-vuejs/blob/master/homework/chapter3.html

4. List Rendering

In the fourth chapter of this book, we are going to learn about list rendering. Using Vue's directives we are going to demonstrate how to:

1. Render a list of items based on an array.
2. Repeat a template.
3. Iterate through the properties of an object.
4. Filter an array of items.
5. Order an array of items.
6. Apply a custom filter to a list.

4.1 Install & Use Bootstrap

To make our work easier on the eye, we are going to import Bootstrap.

Info

Bootstrap is the most popular HTML, CSS, and JS framework for developing responsive, mobile first projects on the web.

Head to http://getbootstrap.com/[1] and click the download button. For the time being, we'll just use Bootstrap from the CDN link[2] but you can install it in any way that suits your particular needs. For our example we need only one file, for now: **`css/bootstrap.min.css`**. When we use this **`.css`** file in our app, we have access to all the pretty structures and styles. Just include it within the **`head`** tag of your page and you are good to go.

Bootstrap requires a containing element to wrap site contents and house our grid system. You may choose one of two containers to use in your projects. Note that, due to **`padding`** and more, neither container is nestable.

- Use **`.container`** for a responsive fixed width container.

   ...
- Use **`.container-fluid`** for a full width container, spanning the entire width of your viewport.

   ...

[1] http://getbootstrap.com/
[2] https://www.bootstrapcdn.com/

At this point, we would like to make an example of Vue.js with Bootstrap classes. This is the introductory example concerning classes and many will follow. Of course, not much study or experimentation is required in order to make use of combined Vue and Bootstrap.

```
<html>
<head>
<link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.cs\
s" rel="stylesheet">
    <title>Hello Bootstrap</title>
</head>
<body>
    <div class="container">
        <h1>Hello Bootstrap, sit next to Vue.</h1>
        <pre>
            {{$data | json}}
        </pre>
    </div>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
<script type="text/javascript">
    new Vue({
        el: '.container',
        data: {

        }
    })
</script>
</html>
```

What you can see here, is the installed Bootstrap and the basic set up for our stories example.

Notice this time, instead of targeting **app** id, we have targeted the **container** class within the **el** option inside the Vue instance. Going that way, we have gained the styles and structure that come along with this class and made our app a bit more delightful.

Note

Most of the times we are going to use the **pre** tag in our code to display our data in JSON format.

Tip

In the above example we target the element with class of `.container`. **Be careful** when you are targeting an element by class, when the class is present more than 1 time, Vue.js will mount on the first element **only**.

Using `el:` you can target any DOM element! Try targeting the `body` of your HTML and see how that works!

4.2 v-for

In order to loop through each item in an array, we will use **v-for** Vue's directive.

The **v-for** loop works on arrays/objects and is used to loop through each item in an array. This directive requires a special syntax in the form of **item in array** where **array** is the source data Array and **item** is an alias for the Array element being iterated on.

Warning

If you are coming from the php world you may notice that **v-for** is similar to php's **foreach** function. But **be careful** if you are used to **foreach($array as $value)**.

Vue's **v-for** is exactly the opposite, **value in array**.

The singular first, the plural next.

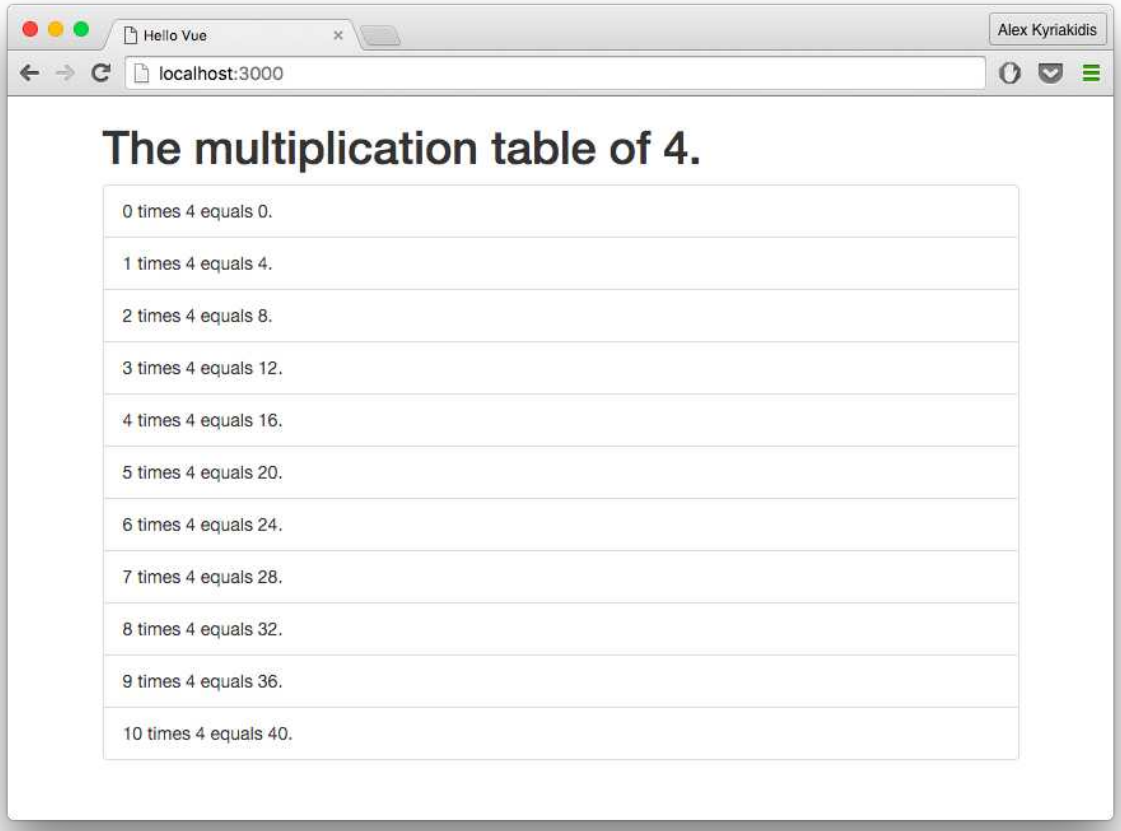
4.2.1 Range v-for

Directive **v-for** can also take an integer. Whenever a number is passed instead of an array/object, the template will be repeated as many times as the number given.

```
<html>
<head>
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.mi\
n.css" rel="stylesheet">
    <title>Hello Vue</title>
</head>
<body>
    <div class="container">
        <h1>The multiplication table of 4.</h1>
        <ul class="list-group">
            <li v-for="i in 11" class="list-group-item">
                {{ i }} times 4 equals {{ i * 4 }}.
            </li>
        </ul>
    </div>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
<script type="text/javascript">
    new Vue({
        el: '.container'
    })
```

```
22 </script>
23 </html>
```

The above code displays the multiplication table of 4.



Multiplication Table of 4

Note

Because we want to display all the multiplication table of 4 (until 40) we repeat the template 11 times since the first value `i` takes is `0`.

4.3 Array Rendering

4.3.1 Loop Through an Array

In the next example we will set up the following array of Stories inside our data object and we will display them all, one by one.

```
stories: [
    "I crashed my car today!",
    "Yesterday, someone stole my bag!",
    "Someone ate my chocolate...",
]
```

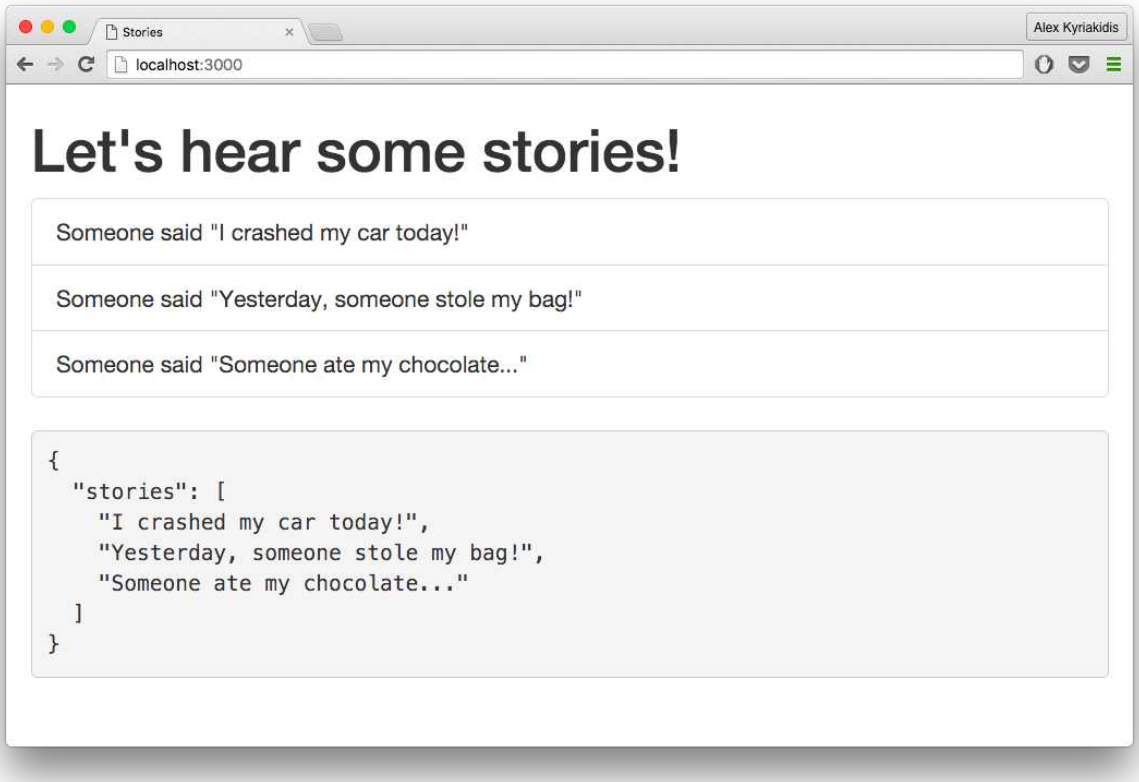
What we need to do here, is to render a list. Specifically, an array of strings.

```
<html>
<head>
<link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.cs\
s" rel="stylesheet">
    <title>Stories</title>
</head>
<body>
    <div class="container">
        <h1>Let's hear some stories!</h1>
        <div>
            <ul class="list-group">
                <li v-for="story in stories" class="list-group-item">
                    Someone said "{{ story }}"
                </li>
            </ul>
        </div>
        <pre>
            {{$data | json}}
        </pre>
    </div>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
<script type="text/javascript">
    new Vue({
        el: '.container',
        data: {
```

```
27              stories: [
28                  "I crashed my car today!",
29                  "Yesterday, someone stole my bag!",
30                  "Someone ate my chocolate...",
31              ]
32          }
33      })
34  </script>
35  </html>
```

Info

Both `list-group` and `list-group-item` classes are Bootstrap classes. Here you can find more information about Bootstrap list styling.[3]



Rendering an array using v-for.

[3] http://getbootstrap.com/css/#type-lists

This is the output of the above code. Using **`v-for`** we have managed to display our stories in a simple unordered list. **It is really that easy!**

4.3.2 Loop Through an Array of Objects

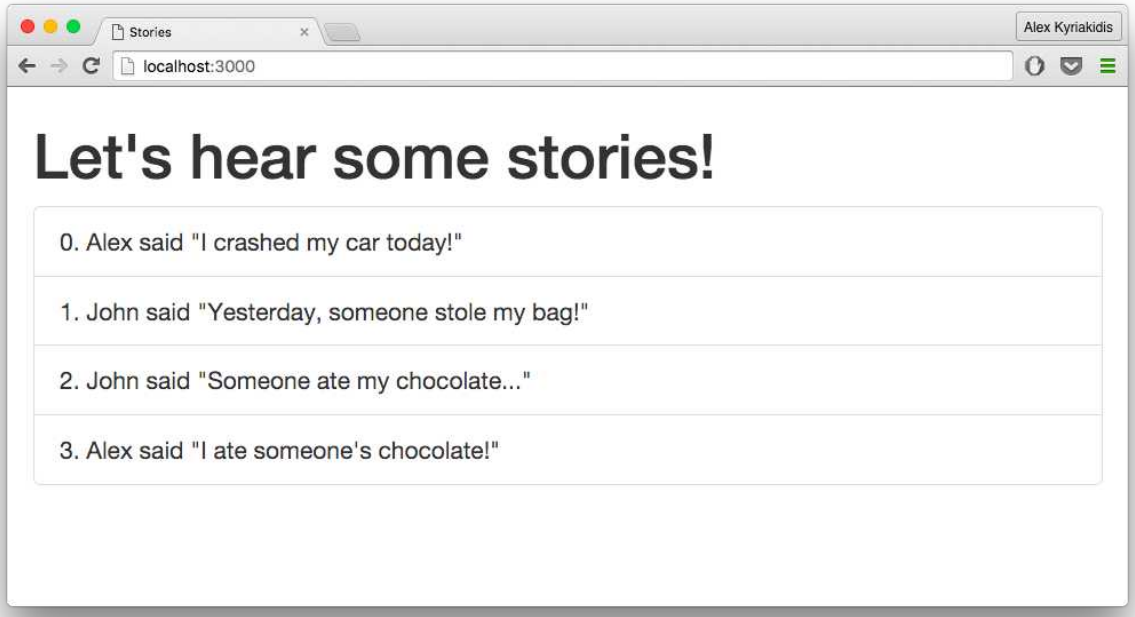
Now, we change the Stories array to contain **`story`** objects. A **`story`** object has 2 properties: **`plot`** and **`writer`**. We will do the same thing we did before but this time instead of echoing **`story`** immediately, we will echo **`story.plot`** and **`story.writer`** respectively.

```
<html>
<head>
<link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.cs\
s" rel="stylesheet">
    <title>Stories</title>
</head>
<body>
    <div class="container">
        <h1>Let's hear some stories!</h1>
        <div>
            <ul class="list-group">
                <li v-for="story in stories"
                    class="list-group-item"
                >
                    {{ story.writer }} said "{{ story.plot }}"
                </li>
            </ul>
        </div>
        <pre>
            {{$data | json}}
        </pre>
    </div>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
<script type="text/javascript">
new Vue({
    el: '.container',
    data: {
        stories: [
            {
                plot: "I crashed my car today!",
                writer: "Alex"
            },
```

```
            {
                plot: "Yesterday, someone stole my bag!",
                writer: "John"
            },
            {
                plot: "Someone ate my chocolate...",
                writer: "John"
            },
            {
                plot: "I ate someone's chocolate!",
                writer: "Alex"
            },
        ]
    }
})
</script>
</html>
```

Additionally, when you need to display the index of the current item, you can use **$index** special variable. Following is an example to show how it works.

```
<ul class="list-group">
    <li v-for="story in stories" class="list-group-item">
        {{$index}}. {{ story.writer }} said "{{ story.plot }}"
    </li>
</ul>
```



Rendered array with index

The **$index** inside the curly braces, clearly represents the index of the iterated item in the given example.

Another way to access the **index** of the iterated item, is to specify an alias for the **index** of the array as it is shown below.

```
<ul class="list-group">
    <li v-for="(index, story) in stories"
        class="list-group-item"
        >
        {{index}} {{ story.writer }} said "{{ story.plot }}"
    </li>
</ul>
```

The output of the last code is exactly the same with the previous one.

4.4 Object v-for

You can use **v-for** to iterate through the properties of an Object. We mentioned before that you can bring to display the **index** of the array, but you can also do the same when iterating an object. In addition to **$index**, each scope will have access to another special property, the **$key**.

Info

When iterating an object, **$index** is in range of **0** ... **n-1** where **n** is the number of object properties.

We have restructured our data to be a single object with 3 attributes this time: **plot**, **writer** and **upvotes**. As you can see in the example code above, we use **$key** and **$index** to bring inside the list the key-value pairs, as well as the **$index** of each pair.

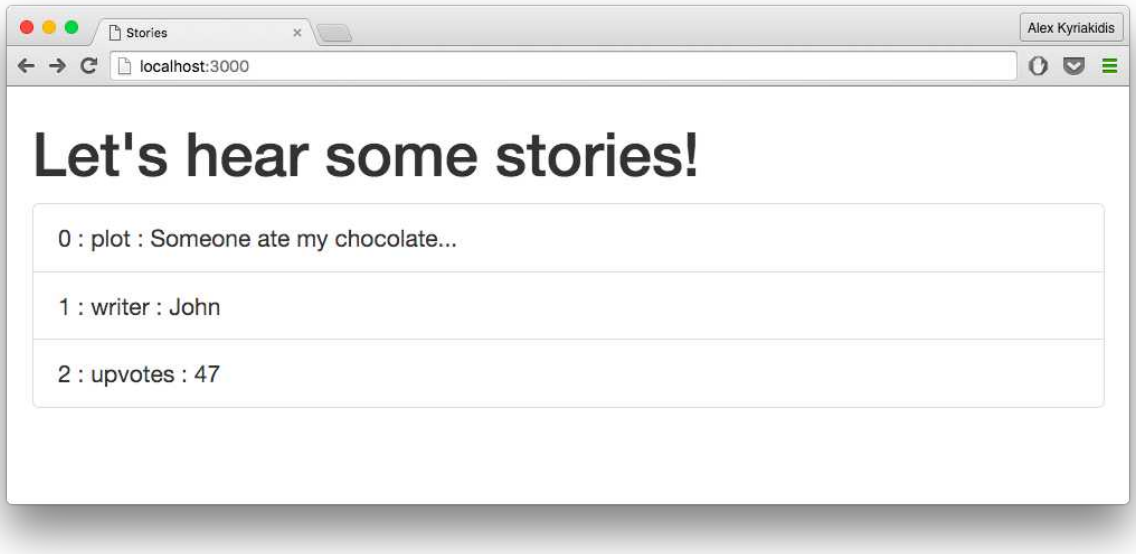
```
<div class="container">
    <h1>Let's hear some stories!</h1>
    <ul class="list-group">
        <li v-for="value in story" class="list-group-item">
            {{$index}} : {{$key}} : {{ value }}
        </li>
    </ul>
</div>
```

```
new Vue({
    el: '.container',
    data: {
        story: {
            plot: "Someone ate my chocolate...",
            writer: 'John',
            upvotes: 47
        }
    }
})
```

Alternatively, you can also specify an alias for the key.

```
<div class="container">
    <h1>Let's hear some stories!</h1>
    <ul class="list-group">
        <li v-for="(key, value) in story"
            class="list-group-item"
            >
           {{$index}} : {{key}} : {{ value }}
        </li>
    </ul>
</div>
```

Either way the result will be:



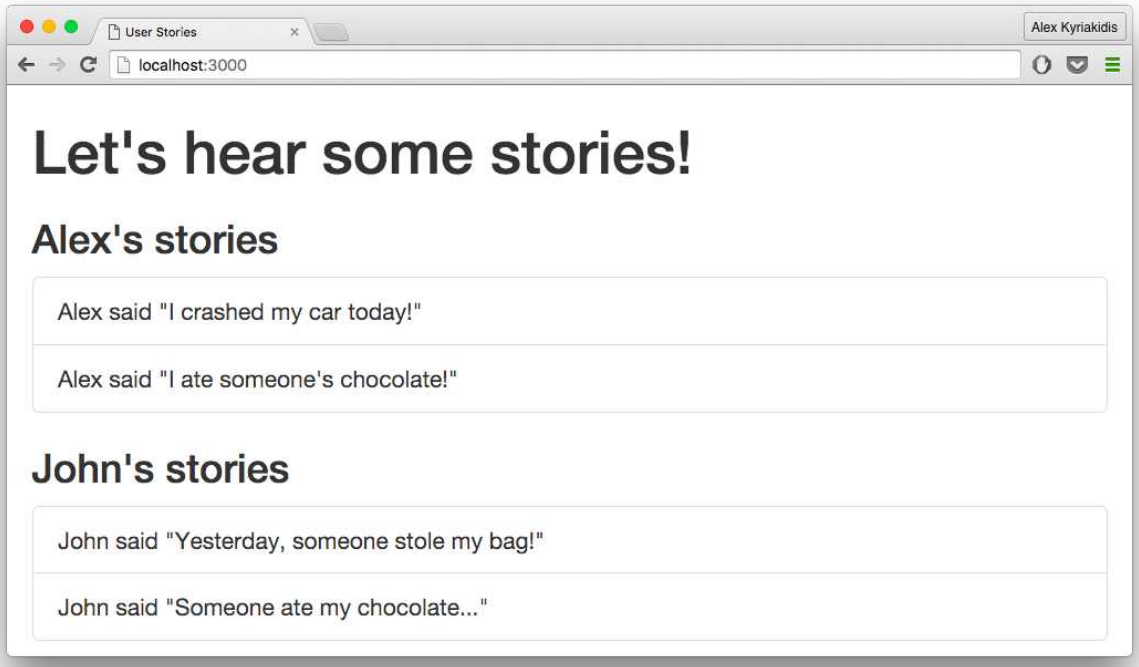
Iterate though object's properties.

4.5 Filtered Results

Sometimes we need to display a filtered version of an array without actually mutating or resetting the original data. In our example we want to display a list with the stories written by Alex and one list with the stories written by John. We can achieve this by using the built-in filter, **`filterBy`**.

```
<html>
<head>
<link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.cs\
s" rel="stylesheet">
    <title>User Stories</title>
</head>
<body>
    <div class="container">
        <h1>Let's hear some stories!</h1>
        <div>
            <h3>Alex's stories</h3>
            <ul class="list-group">
                <li v-for="story in stories | filterBy 'Alex' in 'writer'"
                    class="list-group-item"
                >
                    {{ story.writer }} said "{{ story.plot }}"
                </li>
            </ul>
            <h3>John's stories</h3>
            <ul class="list-group">
                <li v-for="story in stories | filterBy 'John' in 'writer'"
                    class="list-group-item"
                >
                    {{ story.writer }} said "{{ story.plot }}"
                </li>
            </ul>
        </div>
        <pre>
            {{$data | json}}
        </pre>
    </div>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
<script type="text/javascript">
new Vue({
    el: '.container',
```

```
    data: {
        stories: [
            {
                plot: "I crashed my car today!",
                writer: "Alex"
            },
            {
                plot: "Yesterday, someone stole my bag!",
                writer: "John"
            },
            {
                plot: "Someone ate my chocolate...",
                writer: "John"
            },
            {
                plot: "I ate someone's chocolate!",
                writer: "Alex"
            },
        ]
    }
})
</script>
</html>
```



Stories filtered by writer.

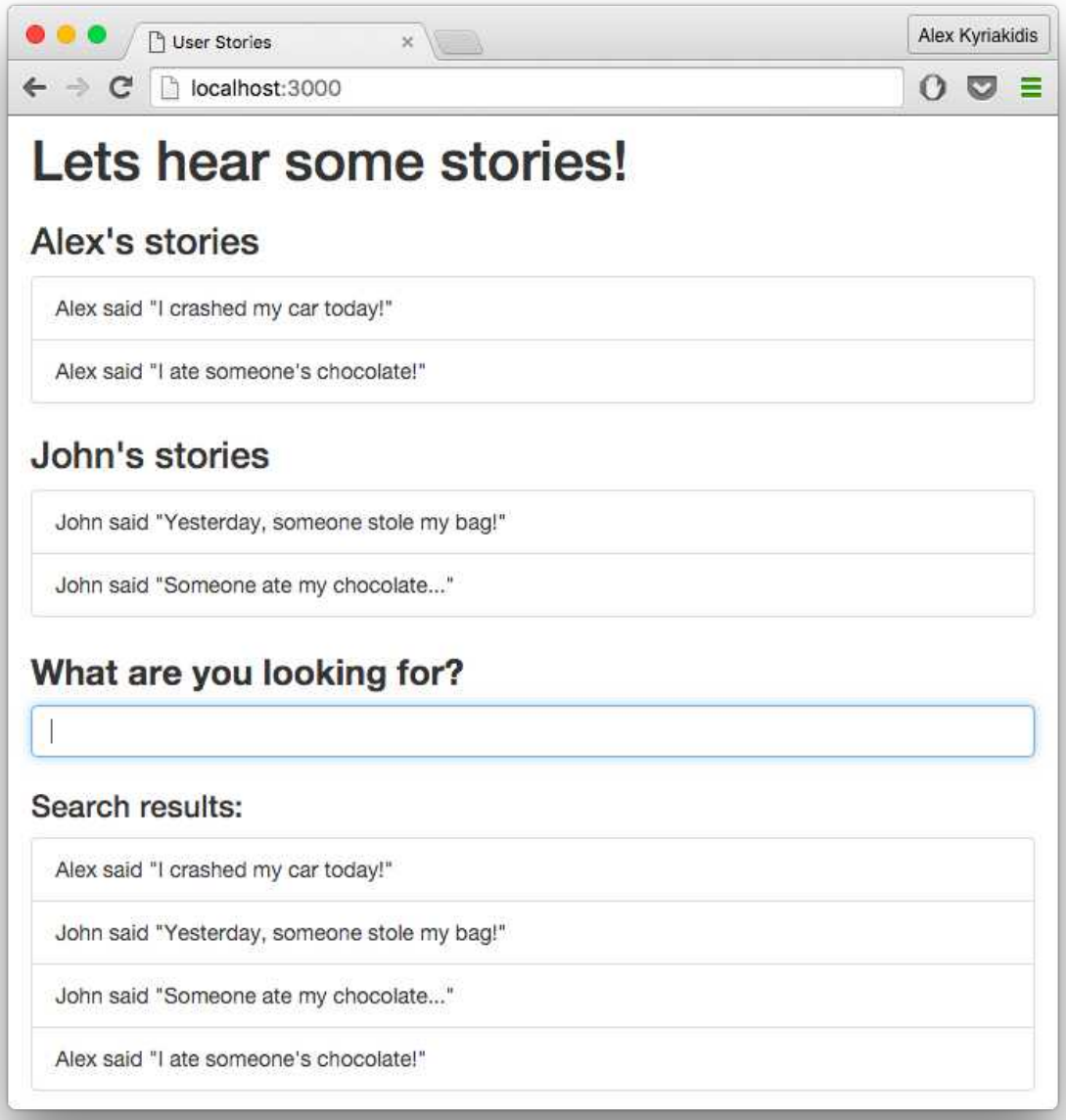
Note

As you may noticed, our `li` tag is getting really big, so we have splitted it in more lines.

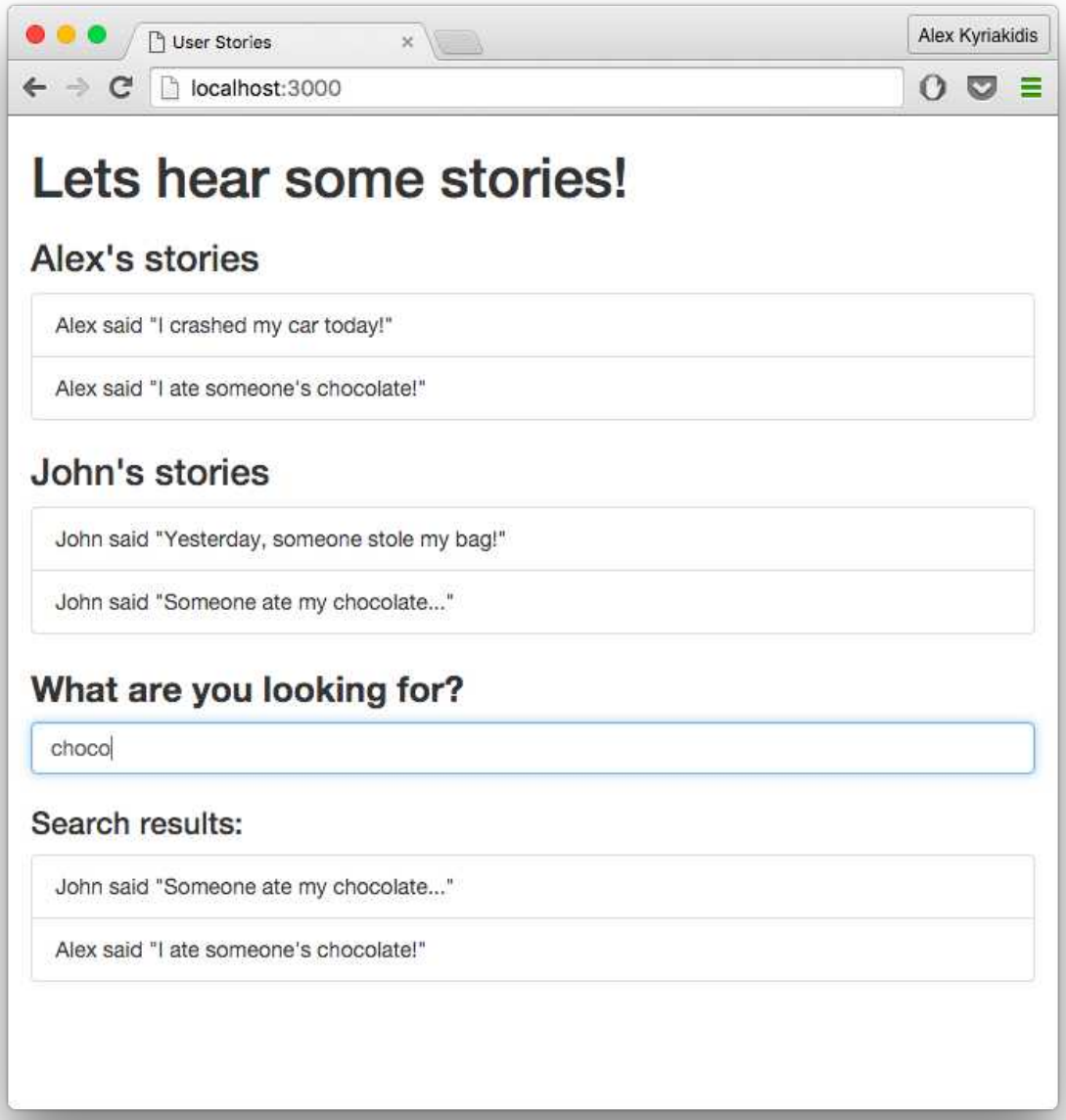
Simple enough, right? Next we will implement a very basic (but awesome) search. When the user types a part of a story, we can guess which story it is and who wrote it, in real time. We'll add a text `input` binded to an empty variable `query` so we can dynamically filter our Stories array.

```
<div class="container">
    <h1>Lets hear some stories!</h1>
    <div>
        <h3>Alex's stories</h3>
        <ul class="list-group">
            <li v-for="story in stories | filterBy 'Alex' in 'writer'"
                class="list-group-item"
            >
                {{ story.writer }} said "{{ story.plot }}"
            </li>
        </ul>
        <h3>John's stories</h3>
```

```
13          <ul class="list-group">
14              <li v-for="story in stories | filterBy 'John' in 'writer'"
15                  class="list-group-item"
16              >
17                  {{ story.writer }} said "{{ story.plot }}"
18              </li>
19          </ul>
20          <div class="form-group">
21              <label for="query">
22                  What are you looking for?
23              </label>
24              <input v-model="query" class="form-control">
25          </div>
26          <h3>Search results:</h3>
27          <ul class="list-group">
28              <li v-for="story in stories | filterBy query in 'plot'"
29                  class="list-group-item"
30              >
31                  {{ story.writer }} said "{{ story.plot }}"
32              </li>
33          </ul>
34      </div>
35  </div>
```



Stories filtered by writer with search.



Searching for 'choco'.

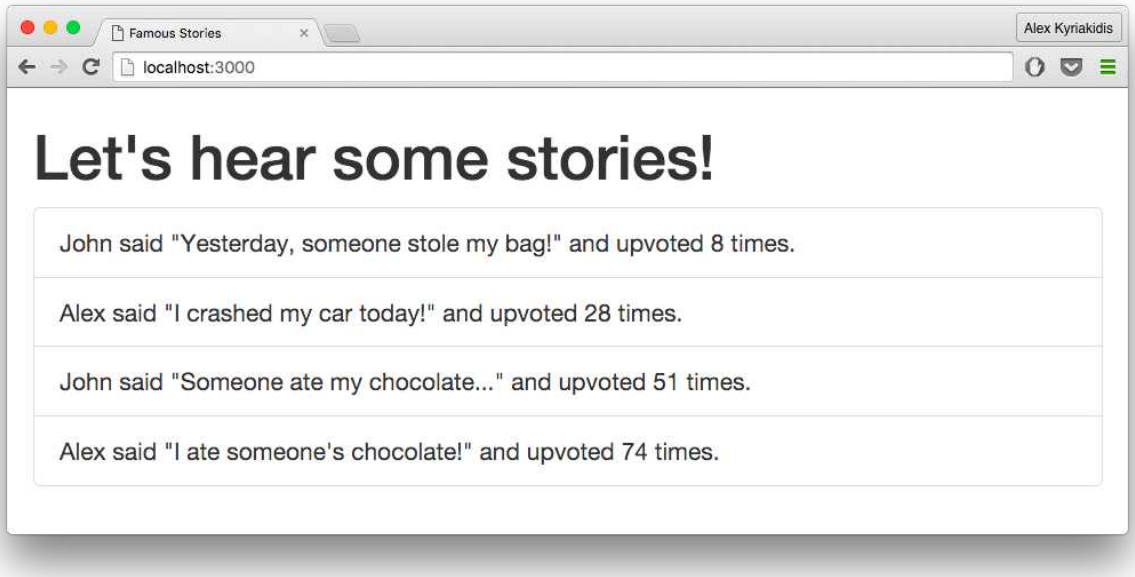
Isn't that awesome??

4.6 Ordered Results

Sometimes we may want to display the items of an Array ordered by some criteria. Luckily, there is an **`orderBy`** built in filter, to sort our list in no time! First we will enhance our Stories with a new property called upvotes. Then we'll go on and display our array ordered by the count of each story's upvotes. *The more famous a story is, the higher it should appear.*

```
<html>
<head>
<link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.cs\
s" rel="stylesheet">
    <title>Famous Stories</title>
</head>
<body>
    <div class="container">
        <h1>Let's hear some stories!</h1>
        <ul class="list-group">
            <li v-for="story in stories | orderBy 'upvotes'"
                class="list-group-item"
            >
                {{ story.writer }} said "{{ story.plot }}"
                and upvoted {{ story.upvotes }} times.
            </li>
        </ul>
    <pre>
        {{ $data | json }}
    </pre>
    </div>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
<script type="text/javascript">
new Vue({
    el: '.container',
    data: {
        stories: [
            {
                plot: "I crashed my car today!",
                writer: "Alex",
                upvotes: 28
            },
            {
                plot: "Yesterday, someone stole my bag!",
```

```
                writer: "John",
                upvotes: 8
            },
            {
                plot: "Someone ate my chocolate...",
                writer: "John",
                upvotes: 51
            },
            {
                plot: "I ate someone's chocolate!",
                writer: "Alex",
                upvotes: 74
            },
        ]
    }
})
</script>
</html>
```



Stories array ordered by upvotes.

Hmmm, the array is ordered but this is not what we expected. We wanted the **famous stories first**. Luckily, again, `orderBy` filter accepts two arguments: the `key` to sort the array, and the `order` which specifies whether the result should be ordered in ascending `(order >= 0)` or descending `(order < 0)` order.

Eventually, for the sake of ordering the array in descending order, our code will look like this:

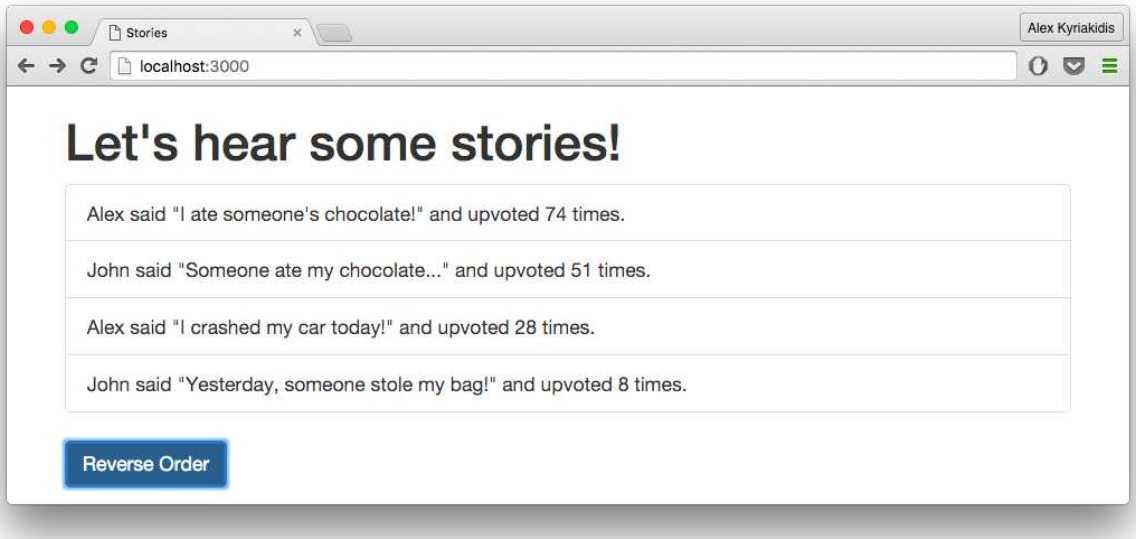
```
<ul class="list-group">
    <li v-for="story in stories | orderBy 'upvotes' -1"
        class="list-group-item"
    >
        {{ story.writer }} said "{{ story.plot }}"
        and upvoted {{ story.upvotes }} times.
    </li>
</ul>
```

We can easily change the order we sort the array, by dynamically changing the **`order`** parameter. A **`button`** is added, which will toggle the value of a new variable between **`-1`** and **`1`**, and then the new variable is passed as **`order`** parameter to **`orderBy`** filter. *Watch now.*

```
new Vue({
    el: '.container',
    data: {
        order: -1,
        stories: [
            {
                plot: "I crashed my car today!",
                writer: "Alex",
                upvotes: 28
            },
            {
                plot: "Yesterday, someone stole my bag!",
                writer: "John",
                upvotes: 8
            },
            {
                plot: "Someone ate my chocolate...",
                writer: "John",
                upvotes: 51
            },
            {
                plot: "I ate someone's chocolate!",
                writer: "Alex",
                upvotes: 74
            },
        ]
    }
})
```

We initialize **order** variable with the value of **-1** and then we pass it to **orderBy** filter.

```
<ul class="list-group">
    <li v-for="story in stories | orderBy 'upvotes' order"
    class="list-group-item"
    >
        {{ story.writer }} said "{{ story.plot }}"
        and upvoted {{ story.upvotes }} times.
    </li>
</ul>
<button @click="order = order * -1">Reverse Order</button>
```



Array in descending order

Impressive huh? If you are not impressed by now, guess who is! .."**We are!**"..

4.7 Custom Filter

This is the most cumbersome part of this chapter. Assume we want to display only the famous stories (the ones with upvotes greater than 20). In order to achieve that we have to create a custom filter and apply it to **`filterBy`**. We are going to create a filter named **`famous`** which expects two parameters:

- the **`array`** we want to filter
- and the **`bound`** which defines the amount of **`upvotes`** a story **must** have, in order to be considered as famous

The **`famous`** filter returns an array which contains only the objects that satisfy a condition.

If you can't keep up with this example don't worry, you will get it sooner or later, just keep reading..

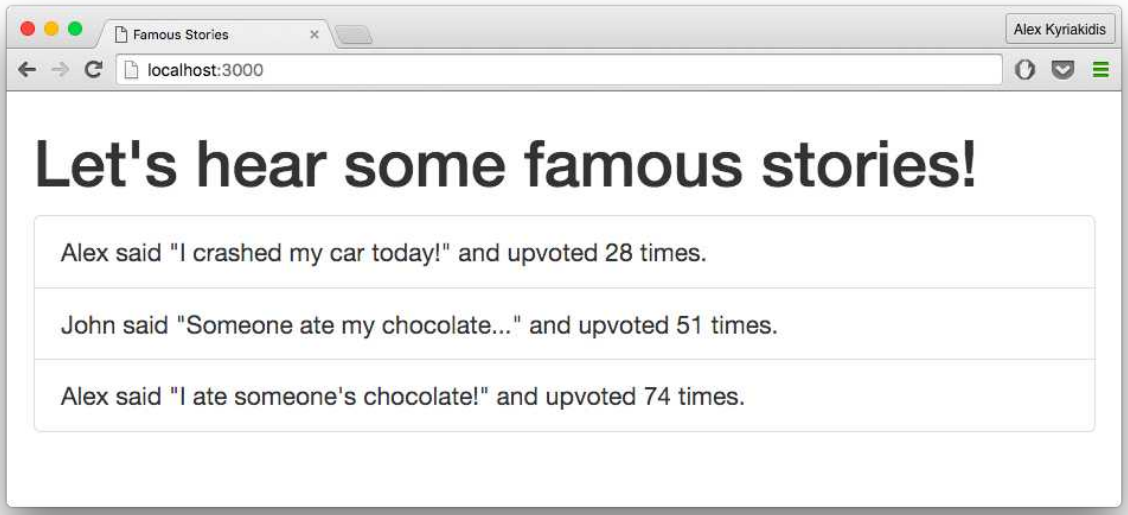
```
<html>
<head>
<link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.cs\
s" rel="stylesheet">
    <title>Famous Stories</title>
</head>
<body>
    <div class="container">
    <h1>Let's hear some famous stories!</h1>
        <ul class="list-group">
            <li v-for="story in stories | famous"
            class="list-group-item"
            >
                {{ story.writer }} said "{{ story.plot }}"
                and upvoted {{ story.upvotes }} times.
            </li>
        </ul>
    <pre>
        {{ $data | json }}
    </pre>
    </div>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
<script type="text/javascript">
Vue.filter('famous', function (stories) {
  return stories.filter(function(item){
    return item.upvotes > 20;
  });
```

```
})

new Vue({
    el: '.container',
    data: {
        stories: [
            {
                plot: "I crashed my car today!",
                writer: "Alex",
                upvotes: 28
            },
            {
                plot: "Yesterday, someone stole my bag!",
                writer: "John",
                upvotes: 8
            },
            {
                plot: "Someone ate my chocolate...",
                writer: "John",
                upvotes: 51
            },
            {
                plot: "I ate someone's chocolate!",
                writer: "Alex",
                upvotes: 74
            },
        ]
    }
})
</script>
</html>
```

Info

Our **famous** filter uses javascript's **filter** method.[4] The **filter()** method creates a new array with all elements that pass the test implemented by the provided function.

[4] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/filter



Custom filter 'famous' in action.

Code Examples

You can find the code examples of this chapter on GitHub[5].

[5] https://github.com/hootlex/the-majesty-of-vuejs/tree/master/examples/4.%20List%20Rendering

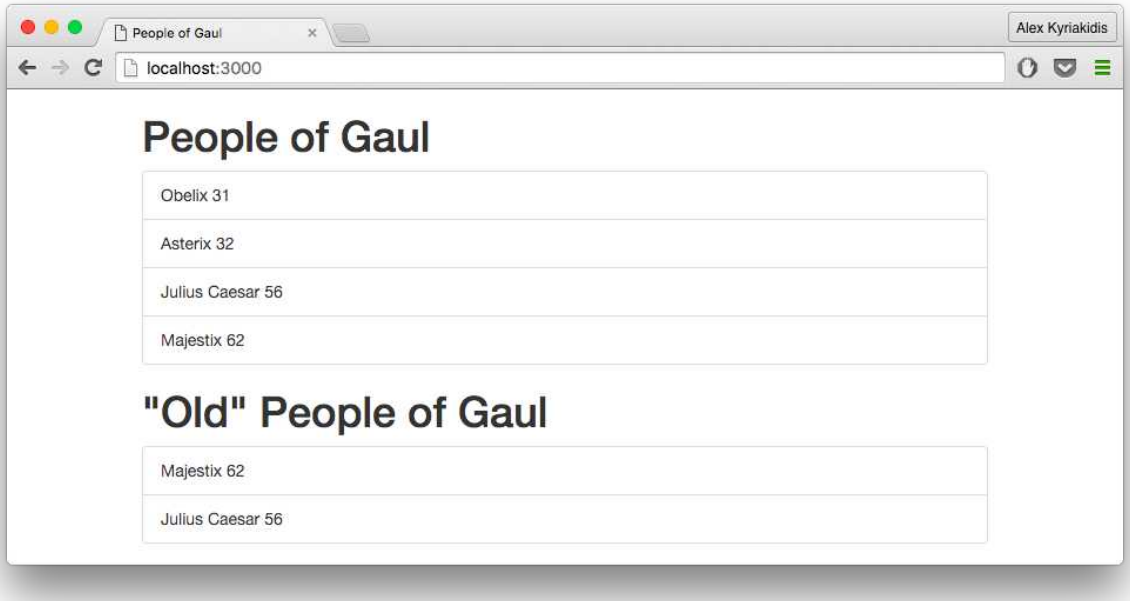
4.8 Homework

For this chapter's exercise you should do the following. Start by creating an array of people. Each person has a name and an age. Using what you've learned so far, try to render the array in a list and sort it by "age". After that, create a second list below and apply a custom filter called "old" which returns all people older than 55 years old.

Feel free to fill the array with your own data. **Be careful** to add people with age older and younger than 55 to ensure your filter is working properly. Go ahead!

Hint

Built in filter `orderBy` and `Vue.filter` are necessary here.



Example Output

You can find a potential solution to this exercise here[6].

[6] https://github.com/hootlex/the-majesty-of-vuejs/blob/master/homework/chapter4.html

5. Interactivity

In this chapter, we are going to create and expand previous examples, learn new things concerning 'methods', 'event handling' and 'computed properties'. We will develop a few examples using different approaches. It's time to see how we can implement Vue's interactivity to get a small app, like a Calculator, running nice and easy.

5.1 Event Handling

HTML events are things that happen to HTML elements. When Vue.js is used in HTML pages, it can **react** to these events.

In HTML, events can represent everything from basic user interactions to things happening in the rendering model.

These are some examples of HTML events:

- A web page has finished loading
- An input field was changed
- A button was clicked
- A form was submitted

The point of event handling is that you can do something whenever an event takes place.

In Vue.js, to **listen** to DOM events you can use the `v-on` directive.

The `v-on` directive attaches an event listener to an element. The type of the event is denoted by the argument, for example `v-on:keyup` listens to the `keyup` event.

Info

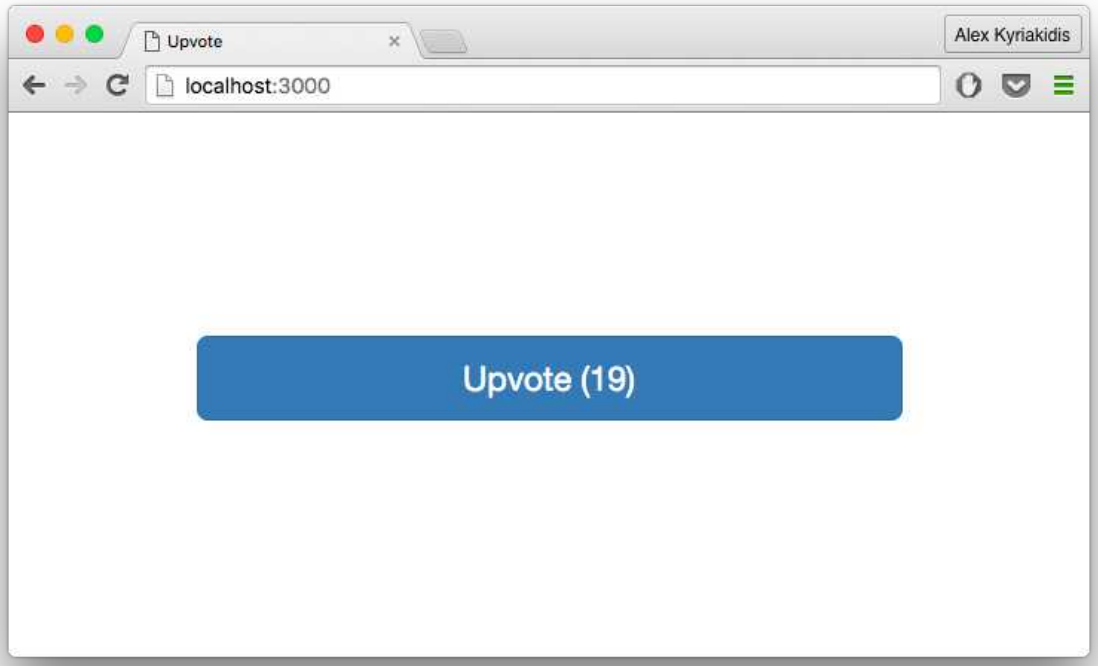
The `keyup` event occurs when the user releases a key. You can find a full list of HTML events here[1].

5.1.1 Handling Events Inline

Enough with the talking, let's move on and see event handling in action. Below, there is an 'Upvote' button which increases the number of upvotes every time it gets clicked.

[1] http://www.w3schools.com/tags/ref_eventattributes.asp

```
<html>
<head>
<link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.cs\
s" rel="stylesheet">
<title>Upvote</title>
</head>
<body>
    <div class="container">
        <button v-on:click="upvotes++">
            Upvote! {{upvotes}}
        </button>
    </div>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
<script type="text/javascript">
new Vue({
    el: '.container',
    data: {
        upvotes: 0
    }
})
</script>
</html>
```



Upvotes counter

As you can see above, we have a basic setup and this time we use the class `container` in our view model. There is an `upvotes` variable within our data. In this case, we bind an event listener for `click`, with the statement that is right next to it. Inside the quotes we're simply increasing the count of upvotes by one, each time the button is pressed, using the increment operator (`upvotes++`).

Shown above is a very simple inline JavaScript statement.

5.1.2 Handling Events using Methods

Now we are going to do the exact same thing as before, using a method instead. A method in Vue.js is a block of code designed to perform a particular task. To execute a method, you have to define it and then invoke it.

```
<html>
<head>
<link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.cs\
s" rel="stylesheet">
<title>Upvote</title>
</head>
<body>
    <div class="container">
        <button v-on:click="upvote">
            Upvote! {{upvotes}}
        </button>
    </div>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
<script type="text/javascript">
new Vue({
    el: '.container',
    data: {
        upvotes: 0
    },
    // define methods under the **`methods`** object
    methods: {
        upvote: function(){
            // **`this`** inside methods points to the Vue instance
            this.upvotes++;
        }
    }
})
</script>
</html>
```

We are binding a click event listener to a method named '**upvote**'. It works just as before, but cleaner and easier to understand when reading your code.

Warning

Event handlers are restricted to execute **one statement only**.

5.1.3 Shorthand for `v-on`

When you find yourself using `v-on` all the time in a project, you will find out that your HTML will quickly becomes dirty. Thankfully, there is a shorthand for `v-on`, the `@` symbol. The `@` replaces the

entire `v-on:` and when using it, the code looks *a lot cleaner*, but everyone has their own practices and this is totally optional.

Using the shorthand, the button of our previous example will be:

Listening to 'click' using `v-on:`

```
<button v-on:click="upvote">
    Upvote! {{upvotes}}
</button>
```

Listening to 'click' using `@` shorthand

```
<button @click="upvote">
    Upvote! {{upvotes}}
</button>
```

5.2 Event Modifiers

Now we will move on and create a Calculator app. To do so, we'll use a form with two inputs and one dropdown to select the desired operation.

Even though the following code seems fine, our calculator does not work as expected.

```
<html>
<head>
  <title>Calculator</title>
  <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.\
css" rel="stylesheet">
</head>
<body>
  <div class="container">
    <h1>Type 2 numbers and choose operation.</h1>
    <form class="form-inline">
        <!-- Notice here the special attribute 'number'
        is passed in order to parse inputs as numbers.-->
        <input v-model="a" number class="form-control">
        <select v-model="operator" class="form-control">
           <option selected>+</option>
           <option>-</option>
           <option>*</option>
           <option>/</option>
```

```
19          </select>
20          <!-- Notice here the special attribute 'number'
21          is passed in order to parse inputs as numbers.-->
22          <input v-model="b" number class="form-control">
23          <button type="submit" @click="calculate"
24          class="btn btn-primary">
25              Calculate
26          </button>
27      </form>
28      <h2>Result: {{a}}  {{operator}}  {{b}} = {{c}}</h2>
29      <pre>
30          {{$data | json}}
31      </pre>
32  </div>
33  </body>
34  <script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
35  <script type="text/javascript">
36    new Vue({
37      el: '.container',
38      data: {
39        a: 1,
40        b: 2,
41        c: null,
42        operator: " ",
43    },
44    methods:{
45        calculate: function(){
46          switch (this.operator) {
47            case "+":
48                this.c = this.a + this.b
49                break;
50            case "-":
51                this.c = this.a - this.b
52                break;
53            case "*":
54                this.c = this.a * this.b
55                break;
56            case "/":
57                this.c = this.a / this.b
58                break;
59        }
60    }
```

```
},
});
</script>
</html>
```

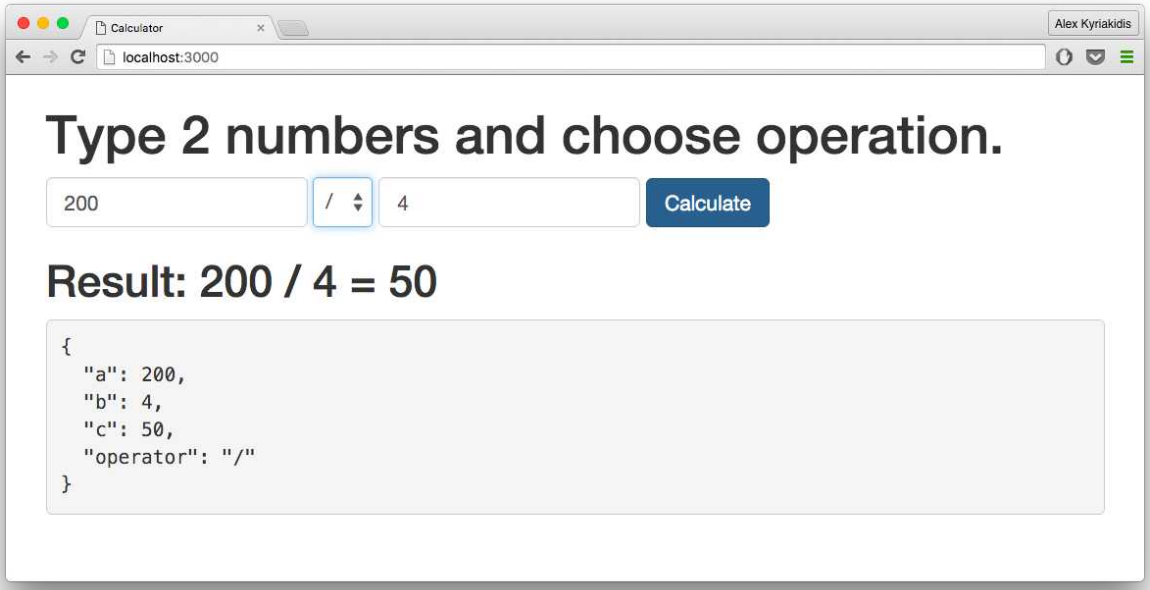
If you try and run this code yourself, you will find out that when the "calculate" button is clicked, instead of calculating, it reloads the page.

This makes sense because when you click "calculate", in the background, you are submitting the form and thus the page reloads.

To prevent the submission of the form, we have to cancel the default action of the **`onsubmit`** event. It is a very common need to call **`event.preventDefault()`** inside our event handling method. In our case the event handling method is called **`calculate`**.

So, our method will become:

```
calculate: function(){
    event.preventDefault();
    switch (this.operator) {
        case "+":
            this.c = this.a + this.b
            break;
        case "-":
            this.c = this.a - this.b
            break;
        case "*":
            this.c = this.a * this.b
            break;
        case "/":
            this.c = this.a / this.b
            break;
    }
}
```



Using Event Modifiers to build a calculator.

Although we can do this easily inside methods, it would be better if the methods can be purely ignorant about data logic rather than having to deal with DOM event details.

Vue.js provides two event modifiers for `v-on` to prevent the event default behavior:

1. `.prevent`
2. `.stop`

So, using one of them, our submit button will change **from**:

```
<button type="submit" @click="calculate">Calculate</button>
```

to:

```
<button type="submit" @click.prevent="calculate">Calculate</button>
<!-- or -->
<button type="submit" @click.stop="calculate">Calculate</button>
```

And we can now safely remove `event.preventDefault()` from our `calculate` method.

5.3 Key Modifiers

If you hit enter when you focus on one of the inputs, you will notice that the page reloads again instead of calculating. This happens because we have prevented the behavior of the submit button but not of the inputs.

To fix this, we have to use '**Key Modifiers**'.

```
<input v-model="a" @keyup.enter="calculate">
<input v-model="b" @keyup.enter="calculate">
```

Tip

When you have a form with a lot of inputs/buttons/etc and you need to prevent their default submit behavior, you can modify the **`submit`** event of the form. Example: <form @submit.prevent="calculate">

Finally, the calculator is up and running.

5.4 Computed Properties

Vue.js inline expressions are very convenient, but for more complicated logic, you should use computed properties. Practically, computed properties are variables which their value depends on other factors.

Computed properties work like functions that you can use as properties. But there is a significant difference. Every time a dependency of a computed property changes, the value of the computed property re-evaluates.

In Vue.js, you define computed properties within the **`computed`** object inside your **`Vue`** instance.

```
<html>
<head>
<link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.cs\
s" rel="stylesheet">
<title>Hello Vue</title>
</head>
<body>
<div class="container">
    a={{ a }}, b={{ b }}
   <pre>
```

```
        {{$data | json}}
    </pre>
</div>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
<script type="text/javascript">
new Vue({
    el: '.container',
    data: {
      a: 1,
    },
    computed: {
      // a computed getter
      b: function () {
        // **`this`** points to the Vue instance
        return this.a + 1
      }
    }
});
</script>
</html>
```

This is a basic example demonstrating the use of computed properties. We've set two variables, the first, **a**, is set to 1 and the second, **b**, will be set by the returned result of the function inside the computed object. In this example the value of **b** will be set to 2.

```
<html>
<head>
<link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.cs\
s" rel="stylesheet">
<title>Hello Vue</title>
</head>
<body>
<div class="container">
    a={{ a }}, b={{ b }}
    <input v-model="a">
    <pre>
        {{$data | json}}
    </pre>
</div>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
```

```
<script type="text/javascript">
new Vue({
    el: '.container',
    data: {
      a: 1,
    },
    computed: {
      // a computed getter
      b: function () {
        // **`this`** points to the vm instance
        return this.a + 1
      }
    }
});
</script>
</html>
```

The above example is the same as the previous one, but with one difference. An input is binded to the `a` variable. The desired outcome would be to change the value of the binded attribute and immediately update the result of `b`. But notice here, that it does not work as we would expect.

If you run this code and enter an input for variable `a` the number 5, you expect that `b` will be set to 6. Sure, but it doesn't, `b` is set to 51.

Why is this happening? Well, as you might have already thought, `b` takes the given value from the input ("a") as a string, and appends the number 1 at the end of it.

One solution to solve this problem is to use the **`parseFloat()` function that parses a string and returns a floating point number**.

```
new Vue({
    el: '.container',
    data: {
      a: 1,
    },
    computed: {
      b: function () {
        return parseFloat(this.a) + 1
      }
    }
});
```

Another option that comes to mind, is to use the **`<input type="number">`** which is used for input fields that should contain a numeric value.

But there is a more neat way. With Vue.js, whenever you want your user's inputs to be automatically persisted as numbers, you can add the special attribute **number** to these inputs.

```
<body>
<div class="container">
    a={{ a }}, b={{ b }}
    <input v-model="a" number>
    <pre>
        {{$data | json}}
    </pre>
</div>
</body>
```

The **number** attribute is going to give us the desired result without any further effort.

To demonstrate a wider picture of computed properties, we are going to make use of them and build the calculator we have already shown, but this time using computed properties instead of methods.

Lets start with a simple example, where a computed property **c** contains the sum of **a** plus **b**.

```
<html>
<head>
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.mi\
n.css" rel="stylesheet">
    <title>Hello Vue</title>
</head>
<body>
    <div class="container">
        <h1>Enter 2 numbers to calculate their sum.</h1>
        <form class="form-inline">
            <input v-model="a" number class="form-control">
            +
            <input v-model="b" number class="form-control">
        </form>
        <h2>Result: {{a}} + {{b}} = {{c}}</h2>
        <pre> {{$data | json}} </pre>
    </div>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
<script type="text/javascript">
    new Vue({
      el: '.container',
      data: {
```

```
        a: 1,
        b: 2
    },
    computed: {
    c: function () {
      return this.a + this.b
  }
}
});
</script>
</html>
```

The initial code is ready, and at this point the user can type in 2 numbers and get the sum of these two. A calculator that can do the four basic operations is the goal, so let's continue building!

Since the HTML code will be the same with the calculator we build in the previous section of this chapter (except now we don't need a button), I am am going to show you here only the Javascript codeblock.

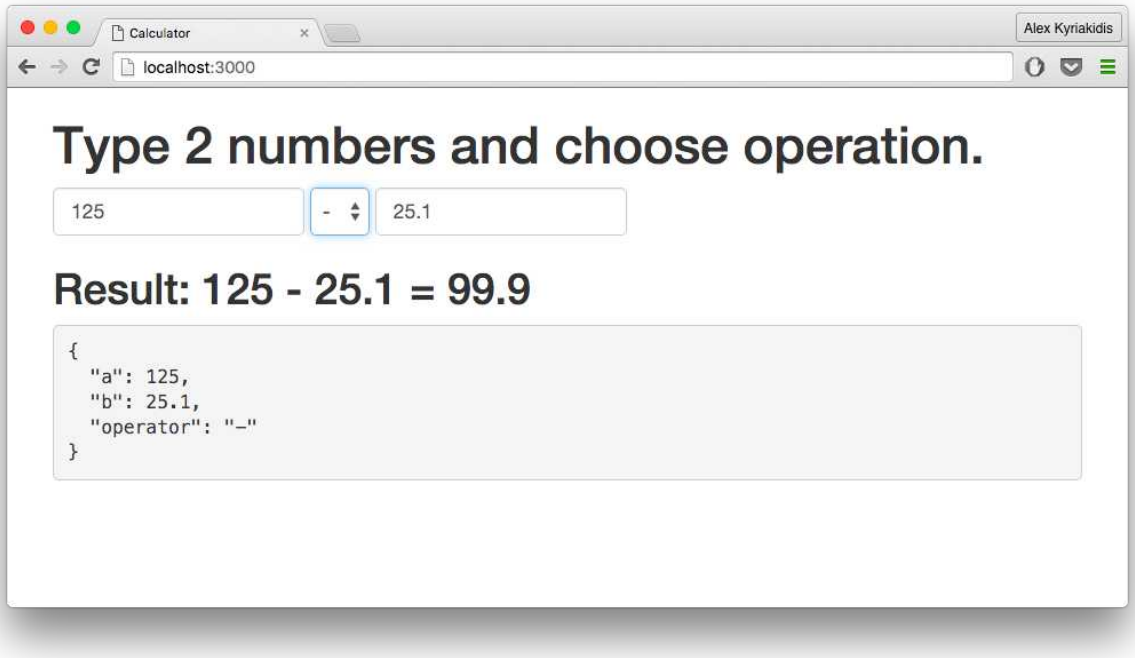
```
new Vue({
        el: '.container',
        data: {
            a: 1,
            b: 2,
            operator: " ",
        },
        computed: {
            c: function () {
                switch (this.operator) {
                    case "+":
                    return this.a + this.b
                    break;
                    case "-":
                    return this.a - this.b
                    break;
                    case "*":
                    return this.a * this.b
                    break;
                    case "/":
                    return this.a / this.b
                    break;
                }
            }
```

```
25            },
26        });
```

The calculator is ready for use. The only thing we had to do, was to move whatever was inside `calculate` method to the computed property `c`! Whenever you change the value of `a` or `b` the result updates in real time! We don't need any buttons, events, or anything. **How awesome is that??**

Info

Note here that a normal approach would be to have an `if` statement to avoid error of division. The best part about this, is that there is already a prediction for this kind of flaws. If the user types 1/0 the result automatically becomes infinity! If the user types a text the displayed result is "not a number".



Calculator built with computed properties.

5.4.1 Using Computed Properties to Filter an Array

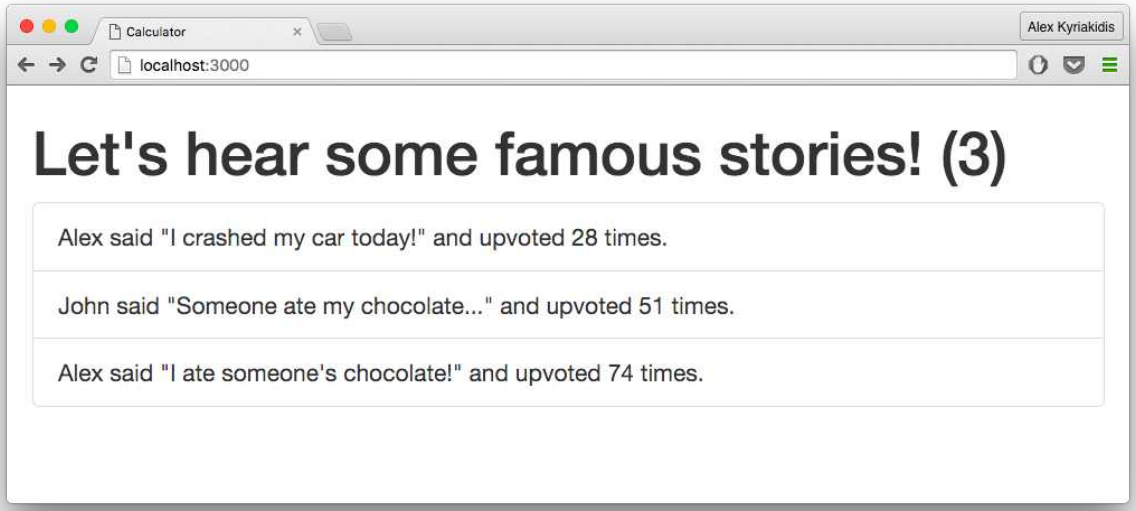
A *computed property* can also be used to filter an array. Using a computed property to perform array filtering gives you in-depth control and more flexibility, since it's full JavaScript, and allows you to access the filtered result elsewhere. For example you can get the length of a filtered array anywhere in your code.

To see how it's done, we will filter the **famous** stories as we did in the Custom Filter example. This time we will create a computed property that returns the filtered Array.

```
new Vue({
    el: '.container',
    data: {
        stories: [
            {
                plot: "I crashed my car today!",
                writer: "Alex",
                upvotes: 28
            },
            {
                plot: "Yesterday, someone stole my bag!",
                writer: "John",
                upvotes: 8
            },
            {
                plot: "Someone ate my chocolate...",
                writer: "John",
                upvotes: 51
            },
            {
                plot: "I ate someone's chocolate!",
                writer: "Alex",
                upvotes: 74
            },
        ]
    },
    computed: {
        famous: function() {
            return this.stories.filter(function(item){
                return item.upvotes > 25;
            });
        }
    }
})
```

In our HTML code, instead of **stories** array, we will render the **famous** computed property.

```
<body>
    <div class="container">
    <h1>Let's hear some famous stories! ({{famous.length}})</h1>
        <ul class="list-group">
            <li v-for="story in famous"
            class="list-group-item"
            >
                {{ story.writer }} said "{{ story.plot }}"
                and upvoted {{ story.upvotes }} times.
            </li>
        </ul>
    </div>
</body>
```



Filter array using a computed property

That's it. We have filtered our array using a computed property. Did you notice how easily we managed to display the *number of famous stories* next to our heading message using `{{famous.length}}`?

Info

Although using a **computed property** to perform array filtering, gives you more flexibility, **array filters** can be more convenient for common use cases.

Code Examples

You can find the code examples of this chapter on GitHub[2].

[2] https://github.com/hootlex/the-majesty-of-vuejs/tree/master/examples/5.%20Interactivity

5.5 Homework

Now that you have a basic understanding of Vue's event handling, methods, computed properties etc, you should try something a bit more challenging. Start by creating an array of "Mayor" candidates. Each candidate has a "name" and a number of "votes". Use a button to increase the count of votes for each candidate. Use a computed property to determine who is the current "Mayor", and display his name.

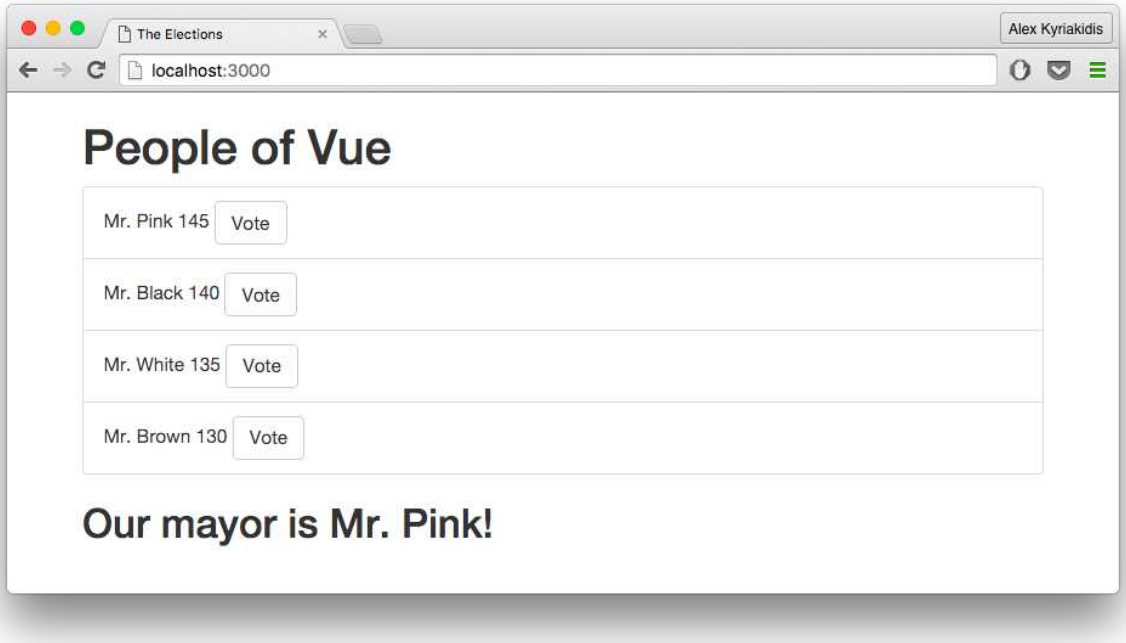
Finally when key '**c**' is pressed the elections start from the beginning, and all votes become 0.

Hint

Javascript's `sort()` and `map()` methods could prove very useful and Key modifiers will get you there.

Hint 2

To listen globally for events you should target the `body` element.



Example Output

You can find a potential solution to this exercise here[3].

[3] https://github.com/hootlex/the-majesty-of-vuejs/blob/master/homework/chapter5.html

6. Components

6.1 What are Components?

Components are one of the most powerful features of Vue.js. They help you extend basic HTML elements to encapsulate reusable code. At a high level, Components are custom elements that Vue.js' compiler would attach specified behavior to. In some cases, they may also appear as a native HTML element extended with the special **`is`** attribute.

It is a really clever and powerful way to extend HTML in order to do new things. In this chapter we are going to start out with an extremely simple example and next we are going to see how Components can help us improve the code we have created, in some of the previous chapters.

6.2 Using Components

We are going to start with a simple Component. In order to use a component we have to register it first.

One way to register a component is to use the **`Vue.component`** method and pass in the **`tag`** and the **`constructor`**. Think of the **`tag`** as the name of the Component and the **`constructor`** as the options. In our occasion, we'll name the Component **`story`** and we'll define the property **`story`** (again). The option **`template`** (how we would like our story to be displayed), is inside the **`constructor`**, where other options will be added as well.

Our story component will be registered like this

```
1  Vue.component('story', {
2      template: '<h1>My horse is amazing!</h1>'
3  });
```

Now that we have registered the component we will make use of it. We will add the custom element **`<story>`** inside the HTML to display the story.

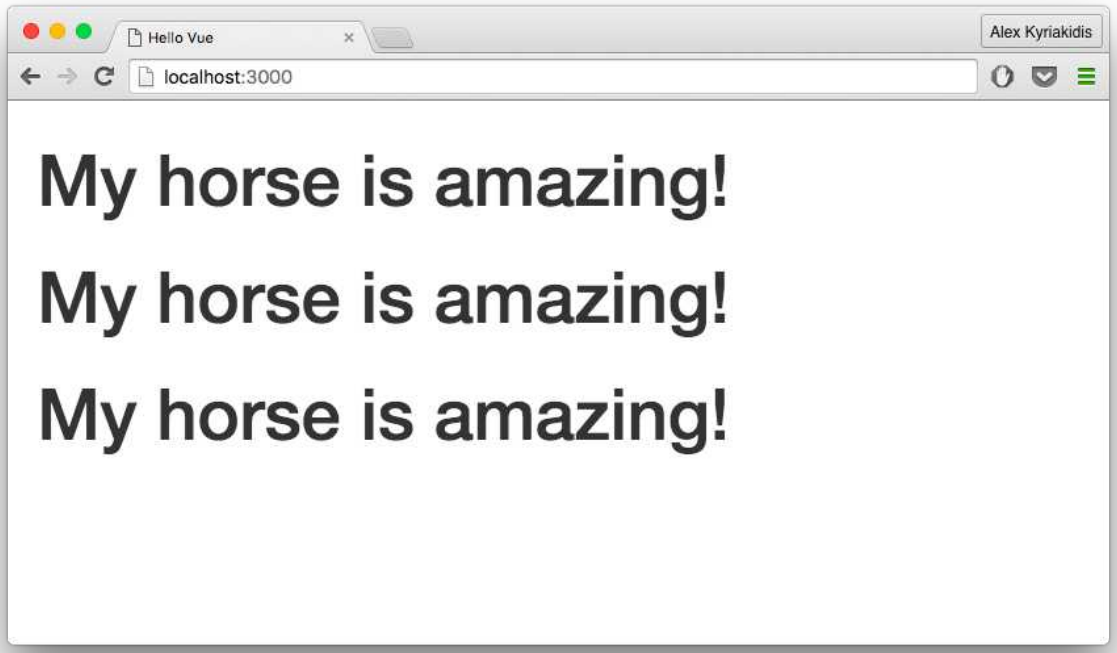
```
<html>
<head>
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.mi\
n.css" rel="stylesheet">
    <title>Hello Vue</title>
</head>
<body>
    <div class="container">
        <story></story>
    </div>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
<script type="text/javascript">
Vue.component('story', {
    template: '<h1>My horse is amazing!</h1>'
});

new Vue({
    el: '.container'
})
</script>
</html>
```

Note

Note here that you can give your custom component any name you want, but it is generally recommended that you should use a unique name to avoid having collisions with actual tags that might get introduced at some point in the future, and saving you time from having to change large amounts of code.

As we mentioned at the beginning of the chapter, components are reusable which means you can append as many **<story>** elements as you want. The following HTML snippet will display our story 3 times.

```
<body>
    <div class="container">
        <story></story>
        <story></story>
        <story></story>
    </div>
</body>
```



Displaying story component

6.3 Templates

There is more than one way of using a template for our component. The inline template we've used before can get "dirty" very fast.

Another way to declare a template is to create a **`script`** tag with type set to **`text/template`** and set an **`id`** of **`story-template`**. To use this template we need to reference a selector in the **`template`** option of our component to this script.

```
<script type="text/template" id="story-template">
    <h1>My horse is amazing!</h1>
</script>
<script type="text/javascript">
    Vue.component('story', {
        template: "#story-template"
    });
</script>
```

Info

"text/template" is not a script that the browser can understand and so the browser will simply ignore it. This allows you to put anything in there, which can then be extracted and generate HTML snippets.

My favorite way to define a **template** (and the one I am going to use in the examples of this book) is to create a **template** HTML tag and give it an **id**. Then we can reference a selector as we did before. Using this technique the above component will look like this:

```
<template id="story-template">
    <h1>My horse is amazing!</h1>
</template>
<script type="text/javascript">
    Vue.component('story', {
        template: "#story-template"
    });
</script>
```

6.4 Properties

Lets see now how we can use multiple instances of our **story** component to display a list of stories. We have to update the **template** to not display always the same story, but the plot of any story we want.

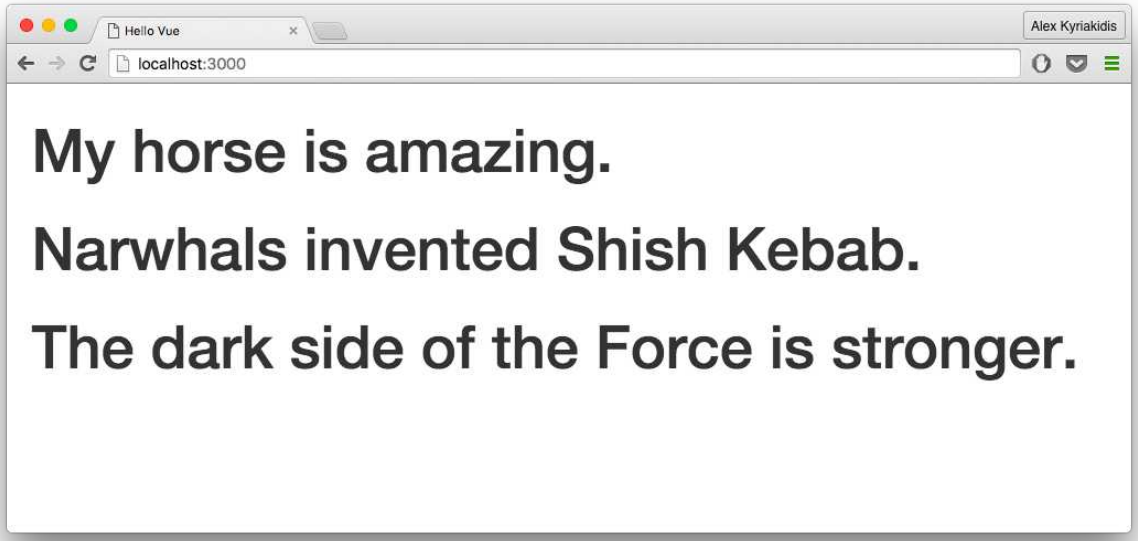
```
<template id="story-template">
    <h1>{{ plot }}</h1>
</template>
```

We also have to update our component to use this property. To do so we will add the new property, 'plot', to **props** attribute of the component.

```
Vue.component('story', {
    props: ['plot'],
    template: "#story-template"
});
```

Now we can pass a **plot** and a plain string to it, every time we use the **<story>** element.

```
<body>
    <div class="container">
        <story plot="My horse is amazing."></story>
        <story plot="Narwhals invented Shish Kebab."></story>
        <story plot="The dark side of the Force is stronger."></story>
    </div>
</body>
```



Display different 'stories'.

Warning

HTML attributes are case-insensitive. When using camelCased prop names as attributes, you need to use their kebab-case (hyphen-delimited) equivalents. So, camelCase in JavaScript, kebab-case in HTML.Example: `props: ['isUser']` , `<story is-user="hello!"></story>`

As you have probably imagined, a component can have more than one property. For example, if we want to display the writer along with the plot for every story, we have to pass the **writer** too.

```
<story plot="My horse is amazing." writer="Mr. Weebl"></story>
```

If you have a lot of properties and your elements are becoming dirty you can pass an object and display its properties.

We will refactor our example one more time to wrap it up.

```
<html>
<head>
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.mi\
n.css" rel="stylesheet">
    <title>Awesome Stories</title>
</head>
<body>
    <div class="container">
        <story v-bind:story="{plot: 'My horse is amazing.', writer: 'Mr. Weebl'}\
">
        </story>
        <story v-bind:story="{plot: 'Narwhals invented Shish Kebab.', writer: 'M\
r. Weebl'}"
        >
        </story>
        <story v-bind:story="{plot: 'The dark side of the Force is stronger.', w\
riter: 'Darth Vader'}"
        >
        </story>
        <template id="story-template">
          <h1>{{ story.writer }} said "{{ story.plot }}"</h1>
        </template>
    </div>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
<script type="text/javascript">
Vue.component('story', {
    props: ['story'],
    template: "#story-template"
});

new Vue({
    el: '.container'
})
</script>
</html>
```

Info

v-bind is used to dynamically bind one or more attributes, or a component prop to an expression.

Since **story** property is not a string but a javascript object instead of **story="..."** we use **v-bind:story="..."** to bind **story** property with the passed object.

The shorthand for **v-bind** is **:**, so from now on we are going to use it like this: **:story="..."**.

6.5 Reusability

Let's take a look again at our **Filtered Results** example. Assume this time we take the **stories** variable data from an external API through an http call. The API developers decide to rename **plot** story property to **body**. So now, we have to go through our code and make the necessary changes.

Info

Later in this book we will cover how we can use **Vue** to make web requests.

```
<div class="container">
    <h1>Lets hear some stories!</h1>
    <div>
        <h3>Alex's stories</h3>
        <ul class="list-group">
            <li v-for="story in stories | filterBy 'Alex' in 'writer'"
                class="list-group-item"
            >
                {{ story.writer }} said "{{ story.plot }}"
                {{ story.writer }} said "{{ story.body }}"
            </li>
        </ul>
        <h3>John's stories</h3>
        <ul class="list-group">
            <li v-for="story in stories | filterBy 'John' in 'writer'"
                class="list-group-item"
            >
                {{ story.writer }} said "{{ story.plot }}"
                {{ story.writer }} said "{{ story.body }}"
            </li>
        </ul>
```

```
22          <div class="form-group">
23              <label for="query">
24                  What are you looking for?
25              </label>
26              <input v-model="query" class="form-control">
27          </div>
28          <h3>Search results:</h3>
29          <ul class="list-group">
30              <li v-for="story in stories | filterBy query in 'body'"
31                  class="list-group-item"
32              >
33                  {{ story.writer }} said "{{ story.plot }}"
34                  {{ story.writer }} said "{{ story.body }}"
35              </li>
36          </ul>
37      </div>
38  </div>
```

Note

In this particular example syntax highlighting is turned off.

As you may have noticed, we had to do the exact same change 3 times and I don't know about you, but I hate repeating myself. If it doesn't seem like a big deal for you, imagine that you may use the above code block in 100 places, what would you do then? Fortunately, 'Vue' provides a solution for that kind of situations, and this solution has a name, **Component**.

Tip

Whenever you find yourself repeating a piece of functionality, the most efficient way to deal with it is to create a dedicated Component.

Luckily we have created a **`story`** Component in the previous example, which displays the writer and the body for a specified story. We can use the custom element **`<story>`** inside our **`HTML`** and pass each story as we did before with **`:story`** tag, but this time we will do it inside **`v-for`** directive.

So our code will be:

```
<div class="container">
    <h1>Lets hear some stories!</h1>
    <div>
        <h3>Alex's stories</h3>
        <ul class="list-group">
            <story v-for="story in stories | filterBy 'Alex' in 'writer'"
            :story="story"></story>
        </ul>
        <h3>John's stories</h3>
        <ul class="list-group">
            <story v-for="story in stories | filterBy 'John' in 'writer'"
            :story="story"></story>
        </ul>
        <div class="form-group">
            <label for="query">What are you looking for?</label>
            <input v-model="query" class="form-control">
        </div>
        <h3>Search results:</h3>
        <ul class="list-group">
            <story v-for="story in stories | filterBy query in 'body'"
            :story="story"></story>
        </ul>
    </div>
</div>
```

If you try to run this code you will get the following warning.

Vue warning

Vue warn: Unknown custom element: <story> - did you register the component correctly? For recursive components, make sure to provide the "name" option.

To fix this we need to register the Component again. This time we have to make some changes to the component's template. We will change **plot** attribute to **body** and **<h1>** tag to **<li>** to suit our needs.

So, the story's template will be:

```
<template id="story-template">
  <li class="list-group-item">
    {{ story.writer }} said "{{ story.body }}"
  </li>
</template>
```

But the component will be the same.

```
Vue.component('story', {
    props: ['story'],
    template: '#story-template'
});
```

If you run the above code you will see for yourself that everything works same as before, but this time with the use of a custom component.

Pretty neat huh?

Warning

Please be responsible. Don't drink and drive.

6.6 Altogether now

Using our newly acquired knowledge we should be able to build something a bit more complex. Taking the structure example from above, we are going to create a voting system for our **stories**, and add a favorite feature. The way to accomplish this, is through methods, directives, and of course, components.

Lets start with the stories setup.

```
<html>
<head>
<link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.cs\
s" rel="stylesheet">
    <title>Hello Vue</title>
</head>
<body>
<div id="app">
    <div class="container">
        <h1>Let's hear some stories!</h1>
        <ul class="list-group">
            <story v-for="story in stories" :story="story"></story>
        </ul>
        <pre>{{ $data | json }}</pre>
    </div>
</div>
<template id="story-template">
    <li  class="list-group-item">
        {{ story.writer }} said "{{ story.plot }}"
    </li>
</template>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
<script type="text/javascript">
Vue.component('story', {
        template: "#story-template",
        props: ['story'],
});

new Vue({
    el: '#app',
    data: {
        stories: [
            {
                plot: 'My horse is amazing.',
                writer: 'Mr. Weebl',
            },
            {
                plot: 'Narwhals invented Shish Kebab.',
                writer: 'Mr. Weebl',
            },
            {
```

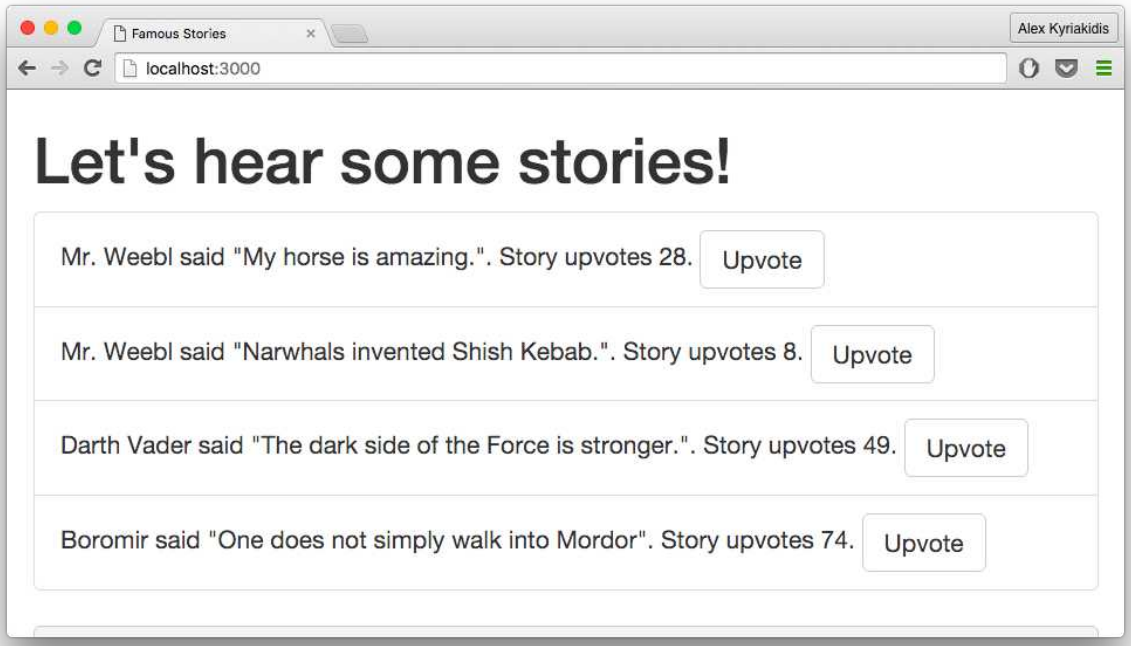
```
                plot: 'The dark side of the Force is stronger.',
                writer: 'Darth Vader',
            },
            {
                plot: 'One does not simply walk into Mordor',
                writer: 'Boromir',
            },
        ]
    }
})
</script>
</html>
```

The next step allows the user to give a vote to the story he prefers. To apply this limit (1 vote per story) we will display the 'Upvote' button only if the user has not already voted. So, every story must have a **voted** property that becomes true when **upvote** function executes.

```
<template id="story-template">
    <li  class="list-group-item">
        {{ story.writer }} said "{{ story.plot }}".
        Story upvotes {{ story.upvotes }}.
        <button v-show="!story.voted" @click="upvote"
                    class="btn btn-default"
        >
            Upvote
        </button>
    </li>
</template>
```

```
Vue.component('story', {
    template: "#story-template",
    props: ['story'],
    methods:{
        upvote: function(){
            this.story.upvotes += 1;
            this.story.voted = true;
        },
    }
});

new Vue({
```

```
    el: '#app',
    data: {
        stories: [
            {
                plot: 'My horse is amazing.',
                writer: 'Mr. Weebl',
                upvotes: 28,
                voted: false,
            },
            {
                plot: 'Narwhals invented Shish Kebab.',
                writer: 'Mr. Weebl',
                upvotes: 8,
                voted: false,
            },
            {
                plot: 'The dark side of the Force is stronger.',
                writer: 'Darth Vader',
                upvotes: 49,
                voted: false,
            },
            {
                plot: 'One does not simply walk into Mordor',
                writer: 'Boromir',
                upvotes: 74,
                voted: false,
            },
        ]
    }
})
```



Ready to vote!

We have implemented, with the use of methods, the voting system. I think it looks good, so we can continue with the 'favorite story' part. We want the user to be able to choose only one story to be his favorite. The first thing that comes to my mind is to add one new empty object (favorite) and whenever the user chooses one story to be his favorite, update **`favorite`** variable. This way we will be able to check if a story is equal to the user's favorite story. Let's do this.

```
<template id="story-template">
    <li  class="list-group-item">
        {{ story.writer }} said "{{ story.plot }}".
        Story upvotes {{ story.upvotes }}.
        <button v-show="!story.voted" @click="upvote"
        class="btn btn-default">
            Upvote
        </button>
        <button v-show="!isFavorite" @click="setFavorite"
        class="btn btn-primary">
            Favorite
        </button>
        <span v-show="isFavorite"
        class="glyphicon glyphicon-star pull-right" aria-hidden="true">
        </span>
```

```
    </li>
</template>
```

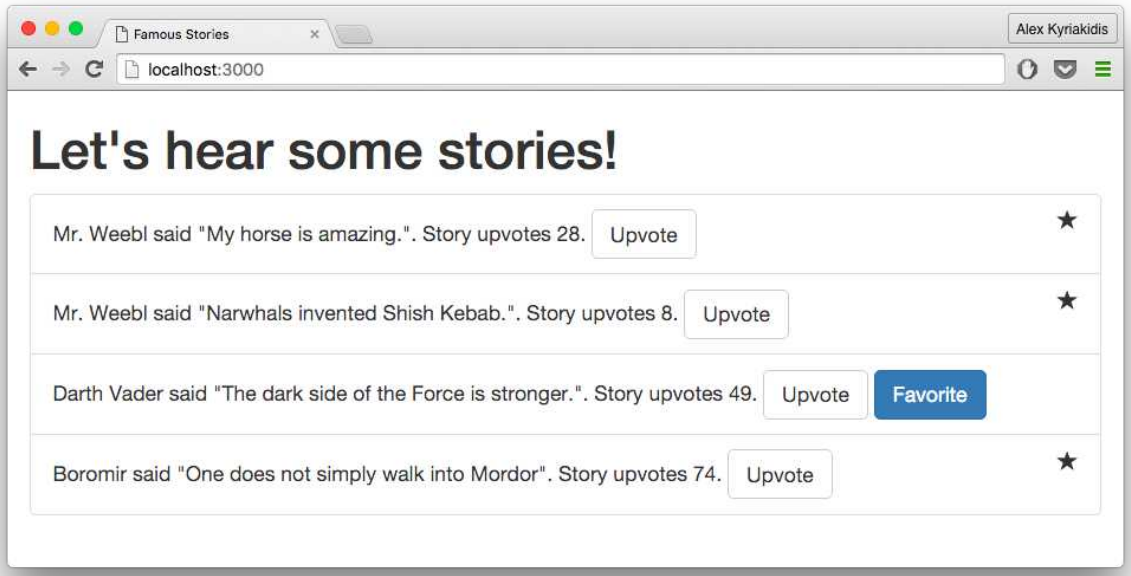
```
Vue.component('story', {
    template: "#story-template",
    props: ['story'],
    methods:{
        upvote: function(){
            this.story.upvotes += 1;
            this.story.voted = true;
        },
        setFavorite: function(){
            this.favorite = this.story;
        },
    },
    computed:{
        isFavorite: function(){
            return this.story == this.favorite;
        },
    }
});

new Vue({
    el: '#app',
    data: {
        stories: [
            ...
        ],
        favorite: {}
    }
})
```

If you try to run the above code, you will notice that it does not work as it should be. Whenever you try to favorite a story, the variable **`favorite`** inside **`$data`** remains null and we get none response.

It seems that our **`story`** component is unable to update **`favorite`** object, so we are going to pass it on each story and add **`favorite`** to component's properties.

```
<ul class="list-group">
    <story v-for="story in stories"
                :story="story"
                :favorite="favorite">
    </story>
</ul>
```

```
Vue.component('story', {
    ...
    props: ['story', 'favorite'],
   ...
});
```

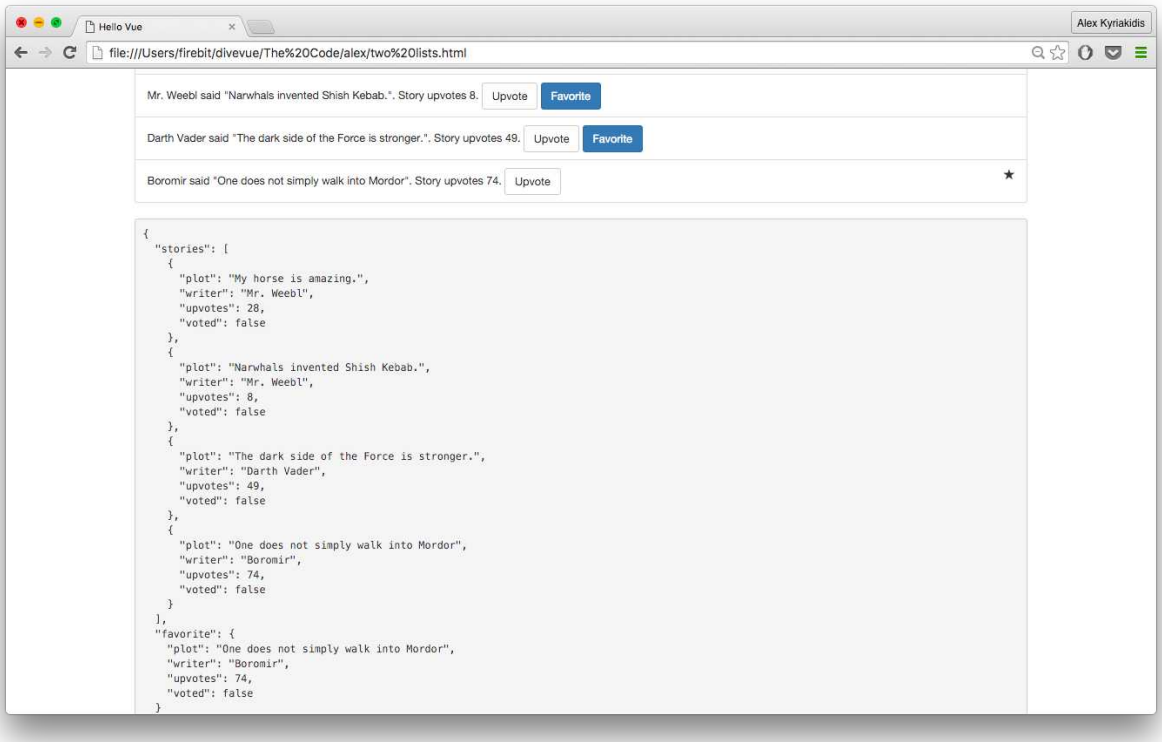


`setFavorite` method malfunctioning

Hmmm, **`favorite`** still doesn't get updated when **`setFavorite`** is executed. The button disappears as expected and a star icon appears, but variable favorite is still null. This results in the user being able to favorite all stories.

The problem with this approach is that we don't keep things *synced*. By default, all props form a one-way-down binding between the child property and the parent one. When the parent property updates, it will flow down to the child, but not the other way around.
This may be confusing but stick with me. In Vue, you can enforce a **two-way binding** with **`.sync`** *binding type modifier*. So, we will pass the variable **`favorite`** to each story like this **`:favorite.sync="favorite"`**.

```
<div id="app">
    <div class="container">
    <h1>Let's hear some stories!</h1>
    <ul class="list-group">
        <story v-for="story in stories"
                  :story="story"
                  :favorite.sync="favorite">
        </story>
    </ul>
    <pre>
        {{ $data | json }}
    </pre>
    </div>
</div>
```



Favorite only one story

Now, the desired result is achieved and the user is able to choose only one story to be his favorite, while he/she can vote as many stories as he/she wants. With the use of **`.sync`** we have synced the property 'favorite' and made the binding two-way with the **`favorite`** object.

Before the `.sync` is a one-way down binding, after is a two-way binding, keeping them

synchronized.

Code Examples

You can find the code examples of this chapter on GitHub[1].

[1] https://github.com/hootlex/the-majesty-of-vuejs/tree/master/examples/6.%20Components

6.7 Homework

This is the most difficult exercise so far, so make sure to put in use everything you have learned in this book. Create an array of 4 horse-drawn chariots. Each chariot has a "name" and a number of "horses" (from 1 to 4). Create a component named "chariot". The "chariot" component should display the name of the chariot and the number of the horses it has. It also must have an action button. The button's text depends on the currently selected chariot.

More specifically, button's text should be:

- 'Pick Chariot', before the user has chosen any chariot
- 'Dismiss Horses', when the chariot has less horses than the selected chariot
- 'Hire Horses', when the chariot has more horses than the selected chariot
- 'Riding!', when the chariot is the selected chariot (this button has to be disabled)

The user should be able to pick a chariot and then choose between any chariot he wants to.

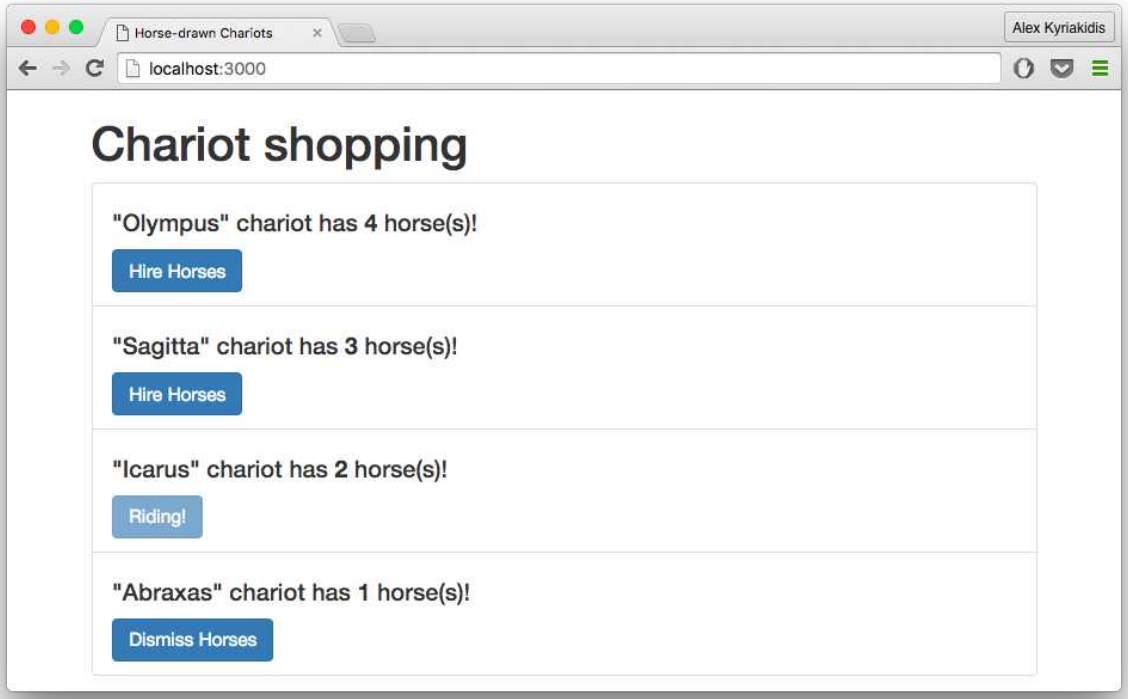
Example Scenario: User has chosen a chariot with **2** horses and its button says 'Riding!'. A chariot with **3** horses has one more horse, so it's button says 'Hire Horses'. A chariot with **1** horse has one less horse than user's chariot, so it's button says 'Dismiss Horses'. I think you got the idea..

Hint

You need to use two-way binding between child's `selectedChariot` property and parent's one.

Hint

To disable a button use `disabled="true"` attribute. You have to figure out how to apply it conditionally.



Example Output

You can find a potential solution to this exercise here[2].

[2] https://github.com/hootlex/the-majesty-of-vuejs/blob/master/homework/chapter6.html

7. Class and Style Bindings

7.1 Class binding

7.1.1 Object Syntax

A common need for data binding is manipulating an element's class and its styles. For such cases, you can use **`v-bind:class`**. This can be used to apply classes conditionally, toggle them and/or apply many of them using one binded object et al.

The **`v-bind:class`** directive takes an object with the following format as an argument

```
{
    'classA': true,
    'classB': false,
    'classC': true
}
```

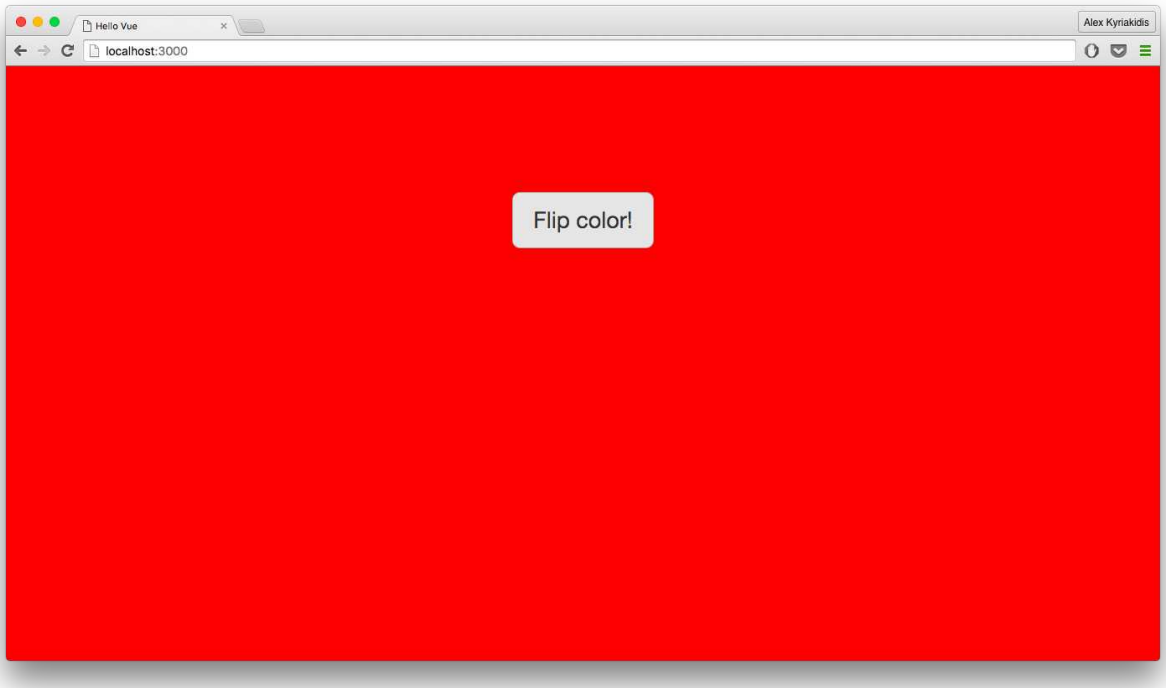
and applies all classes with **`true`** value to the element. For example, the classes of the following element will be **`classA`** and **`classC`**.

```
<div v-bind:class="elClasses"></div>
```

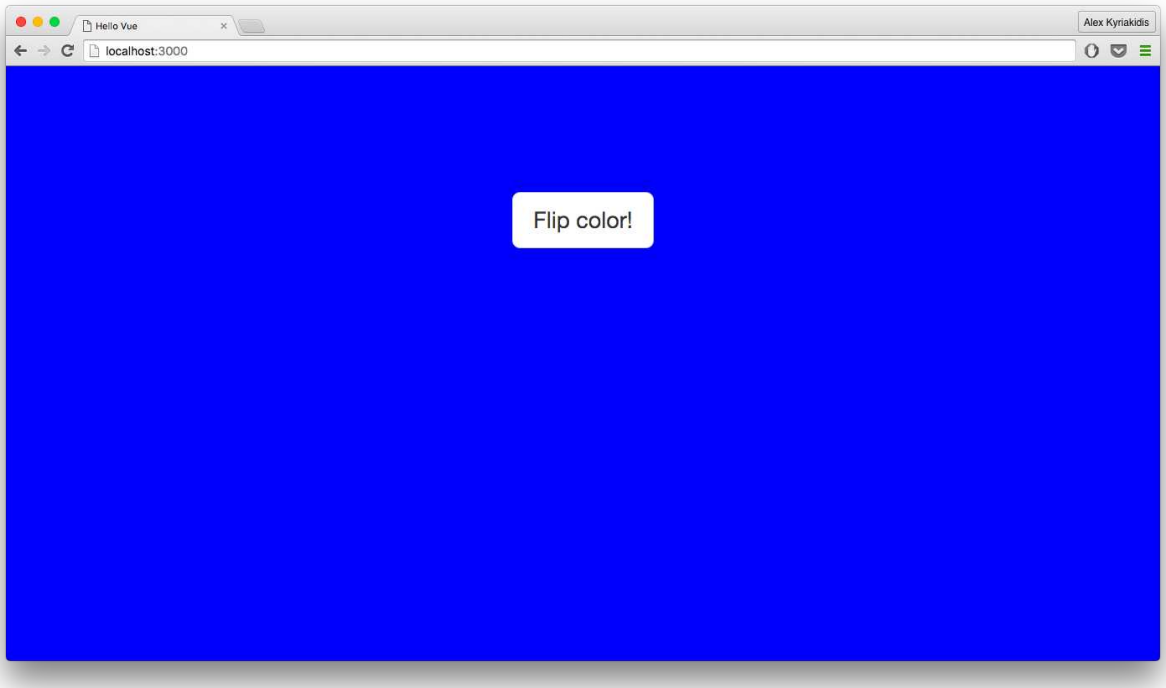
```
data: {
    elClasses:
        {
            'classA': true,
            'classB': false,
            'classC': true
        }
}
```

To demonstrate how **`v-bind`** is used with class attributes, we are going to make an example of class toggling. Using **`v-bind:class`** directive, we are going to dynamically toggle the class of **`body`** element.

```
<html>
<head>
<link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.cs\
s" rel="stylesheet">
<title>Hello Vue</title>
</head>
<body class="text-center"
v-bind:class="{ 'body-red' : color, 'body-blue' : !color }"
>
    <button v-on:click="flipColor" class="btn">
        Flip color!
    </button>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
<script type="text/javascript">
new Vue({
    el: 'body',
    data: {
        color: true
    },
    methods: {
        flipColor: function() {
            this.color = !this.color;
        }
    }
});
</script>
<style type="text/css">
    .body-red {
      background-color: #ff0000;
    }
    .body-blue {
      background-color: #0000ff;
    }
</style>
</html>
```



Changed backround color



Changed backround color

We have applied a class to the `body` for our convenience and now `body` is referenced within our `el` property. What this code actually does, is "flipping" the background color with a hit of the button. Pressing it invokes the `flipColor` function that reverses the value of "color" originally set to `true`. Then the `v-bind:class` is going to toggle the class name to 'body-red' or 'body-blue' conditionally depending on the truthfulness of "color" value. That given, the style is going to apply on each class and give us the desired result, according to which class is active.

Info

The `v-bind:class` directive can co-exist with the plain class attribute.

So in our example, `body` always has the `text-center` class and conditionally one of `body-red` or `body-blue`.

Warning

Although you can use mustache interpolations such as class="{{ className }}" to bind the class, it is not recommended to mix that style with v-bind:class. Use one or the other!

7.1.2 Array Syntax

We can also apply a list of classes to an element, using an array of classnames.

```
<div v-bind:class="['classA', 'classsB', anotherClass]"></div>
```

Applying conditionally a class, can also be achieved with the use of inline `if` inside the array.

```
<div v-bind:class="['classA', condition ? 'classsB' : '']"></div>
```

Info

Inline if is commonly referred to as the **ternary operator**, **conditional operator**, or **ternary if**.

The conditional (ternary) operator is the only JavaScript operator that takes three operands.

The syntax of **ternary operator** is **`condition ? expression1 : expression2`**. If condition is true, the operator returns the value of expression1, otherwise, it returns the value of expression2.

Using inline **`if`**, the flipping colors example will look like:

```
<body class="text-center body"
v-bind:class="[ color ? 'body-red' : 'body-blue']">
    <button v-on:click="flipColor"
    class="btn">
        Flip color!
    </button>
</body>
```

```
new Vue({
  el: 'body',
  data: {
    color: true
  },
  methods: {
    flipColor: function() {
      this.color = !this.color;
    }
  }
});
```

Tip

To actually use a class name instead of a variable inside classes array, use single quotes.
`v-bind:class="[ variable, 'classname']"`

7.2 Style binding

7.2.1 Object Syntax

The Object syntax for **`v-bind:style`** is pretty straightforward; it looks almost like CSS, except it's a JavaScript object. We are going to use the shorthand that Vue.js provides for the previously used directive, **`v-bind(:)`**.

```
<!-- shorthand -->
<div :style="niceStyle"></div>
```

```
data: {
    niceStyle:
    {
        color: 'blue',
        fontSize: '20px'
    }
}
```

We can also declare the style properties inside an object **:style=**"..." inline.

```
<div :style="{'color': 'blue', fontSize: '20px'}"></div>
```

We can even **reference variables** inside style object:

```
<!-- Variable 'niceStyle' is the same we used in the previous example -->
<div :style="{'color': niceStyle.color, fontSize: niceStyle.fontSize}">
</div>
```

Style object binding

It is often a good idea to use a style object and bind it, so the template is cleaner.

7.2.2 Array Syntax

Using inline array syntax for **`v-bind:style`**, we are able to apply multiple style objects to the same element, meaning here that every list item is going to have the **`color`** and **`fontsize`** of **`niceStyle`** and the font weight of **`badStyle`**.

```
<!-- shorthand -->
<div :style="[niceStyle, badStyle]"></div>
```

```
data: {
    niceStyle:
    {
        color: 'blue',
        fontSize: '20px'
    }
    badStyle:
    {
        fontweight: 'bold'
    }
}
```

Info for Intermediates

When you use a CSS property that requires vendor prefixes in v-bind:style, for example transform, Vue.js will automatically detect and add appropriate prefixes to the applied styles.

You can find more information about vendor prefixes here[1].

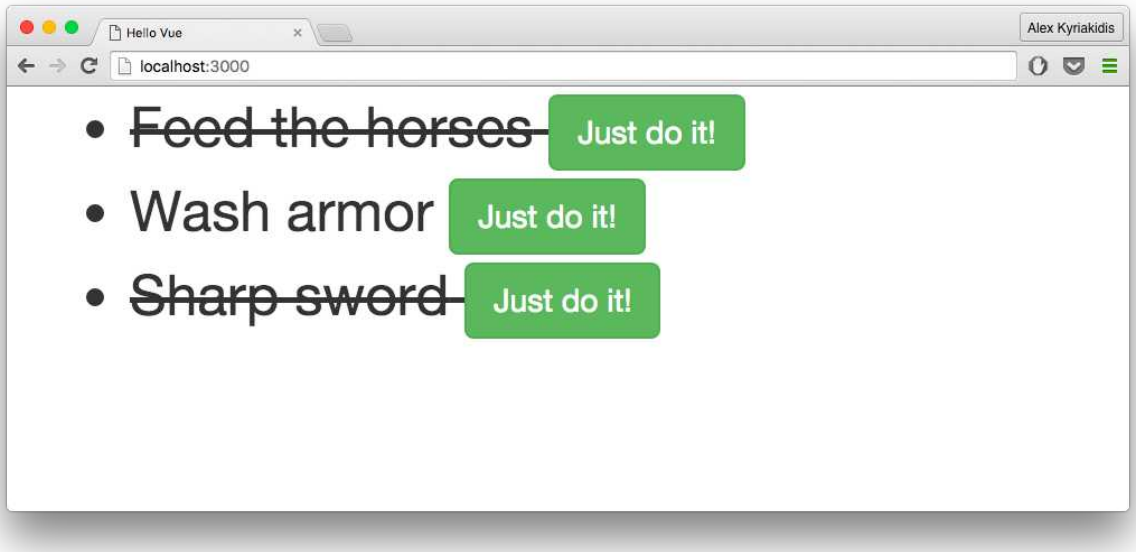
7.3 Bindings in Action

```
<html>
<head>
<link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.cs\
s" rel="stylesheet">
<title>Hello Vue</title>
</head>
<body class="container-fluid">
    <ul>
        <li :class="{'completed' : task.done}"
        :style="styleObject"
        v-for="task in tasks">
            {{task.body}}
        <button @click="completeTask(task)" class="btn">
            Just do it!
        </button>
        </li>
    </ul>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
<script type="text/javascript">
new Vue({
    el: 'body',
    data: {
        tasks: [
                {body: "Feed the horses", done: true},
                {body: "Wash armor", done: true},
                {body: "Sharp sword", done: false},
            ],
        styleObject: {
             fontSize: '25px'
            }
```

[1] https://developer.mozilla.org/en-US/docs/Glossary/Vendor_Prefix

```
    },
    methods: {
        completeTask: function(task) {
            task.done = !task.done;
        }
    },
});
</script>
<style type="text/css">
    .completed {
        text-decoration: line-through;
    }
</style>
</html>
```

The above example has an array of objects called “tasks” and a styleObject which contains only one property. With the use of **`v-for`**, a list of tasks is rendered and each task has a “done” property with a boolean value. Depending on the value of “done”, a class is applied as conditionally as before. If a task has been completed, then **`css`** style applies and the task has a text-decoration of line-through. Each task is accompanied by a button listening for the “click” event which triggers a method, altering the completion status of the task. The **`style`** attribute is binded to **`styleObject`** resulting in the change of ‘fontsize’ of all tasks. As you can see, the **`completedTasks`** method takes in the parameter **`task`**.



Styling completed tasks

Code Examples

You can find the code examples of this chapter on GitHub[2].

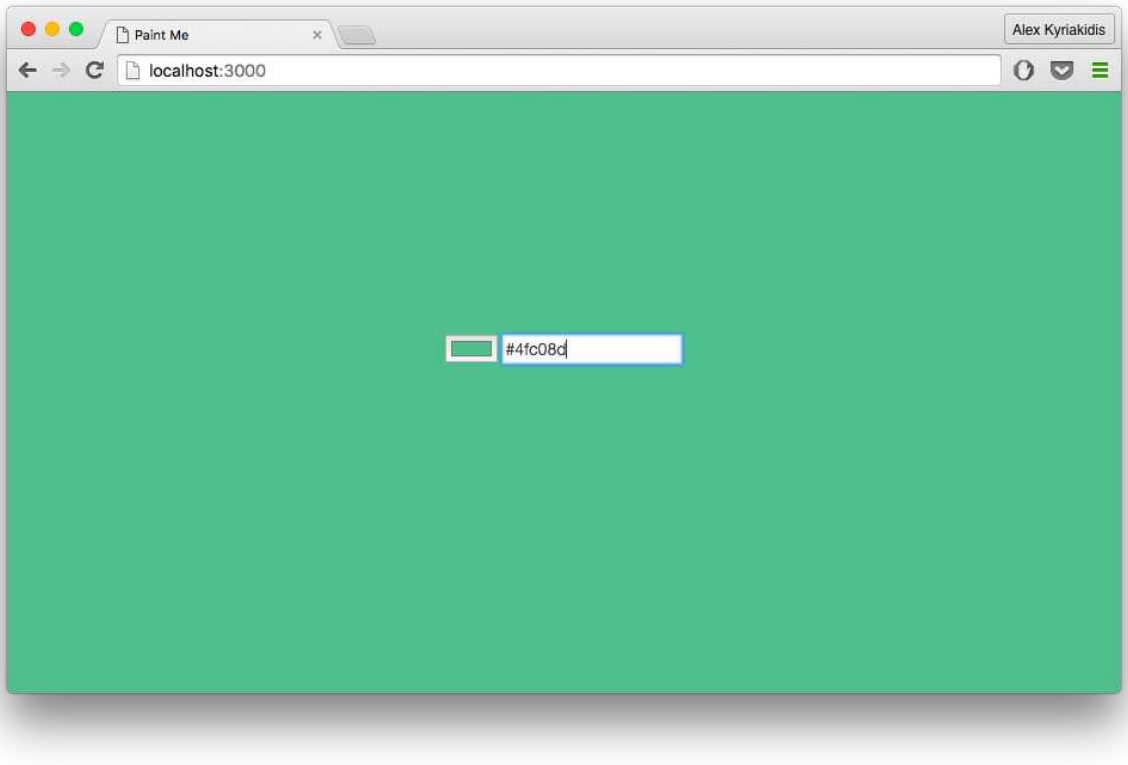
[2] https://github.com/hootlex/the-majesty-of-vuejs/tree/master/examples/7.%20Class%20and%20Style%20Bindings

7.4 Homework

A more fun but maybe tricky exercise for this chapter. Create an input where the user can choose a color. When a color is chosen, apply it to the `body`. Thats it, **let's paint!! :)**

Hint

You could use `input type="color"` for your ease (supported in most browsers).



Example Output

You can find a potential solution to this exercise here[3].

[3] https://github.com/hootlex/the-majesty-of-vuejs/blob/master/homework/chapter7.html

Consuming an API

8. Preface

In this chapter, we are going a little deeper and demonstrate how we can use Vue.js to consume an API.

Following the *story* examples of previous chapters, we are now going to use some real data, coming from an external source.

In order to use real data, we need to make use of a database. Assuming that you already know how to create a database, it won't be covered in this book. To work along with the book's examples, we got you covered, we have already created one to be put to use.

8.1 CRUD

Presume we have a database, we need to perform CRUD operations (Create, Read, Update, Delete). More particularly, we want to

- **Create** new stories in the database
- **Read** existing stories
- **Update** existing story's details (such as 'upvotes')
- **Delete** stories that we don't like

Since Vue.js is a Front-end JavaScript framework, it cannot connect to a database directly. To access a database we need a layer between Vue.js and the database. This layer is the API (Application Program Interface).

8.2 API

Because this book is about Vue.js and not about designing APIs, we will provide you a demo API built with Laravel[1]. Laravel is one of most powerful PHP frameworks along with Symfony2, Nette, CodeIgniter, and Yii2. You are free to create your API using *any language or framework* you like. I use Laravel because it is simple, it has a great community, and it is awesome! :)

Therefore, we strongly recommend you to use the *demo API* that we have built exclusively for the examples of this book.

[1] https://laravel.com/

8.2.1 Download Book's Code

To use our API you have to download the book's code and start a server. To do so, follow the instructions below.

1. Open your terminal and create a directory (we will create '~/themajestyofvuejs')

   ```
   $ mkdir ~/themajestyofvuejs
   ```

2. Download the source code from github

   ```
   $ cd ~/themajestyofvuejs
   $ git clone https://github.com/hootlex/the-majesty-of-vuejs .
   ```

   Alternatively, you can visit the repository on github[2] and download the zip file. Then, extract its contents under the created directory.
3. Navigate to the current chapter under 'apis' of the newly created directory.

   ```
   $ cd ~/themajestyofvuejs/apis/stories
   ```

4. Run the installation script

   ```
   $ sh setup.sh
   ```

5. You now have a database filled with dummy data as well as a fully functional server **running on 'http://localhost:3000'!**

If you want to customize the server(host, port, etc), you can make the setup manually. Below is the source code of our script.

Installation Script: setup.sh

```
# navigate to chapter directory
$ cd ~/themajestyofvuejs/apis/stories

# install dependencies
$ composer install

# Create the database
$ touch database/database.sqlite;

# Migrate & Seed
$ php artisan migrate;
```

[2] https://github.com/hootlex/the-majesty-of-vuejs

```
$ php artisan db:seed;

# Start server
$ php artisan serve --port=3000 --host localhost;
```

Great! You now have a *fully functional API* and a database filled with nice stories.

Note

If you are using Vagrant you have to run the server on **host '0.0.0.0'**. Then, you will be able to access your server on Vagrant's box ip.

If, for example, Vagrant's box ip is *192.168.10.10* and you run

```
$ php artisan serve --port=3000 --host 0.0.0.0;
```

you can browse your website on `192.168.10.10:3000`.

If you have downloaded our demo API, you can continue to the next section.

If you chose to create you own API, you have to create a database table to store the `stories`. The following columns must be present.

Column Name	Type
`id`	Integer, Auto Increment
`plot`	String
`writer`	String
`upvotes`	Integer, Unsigned

Don't forget to seed some fake data to follow up with the next examples.

8.2.2 API Endpoints

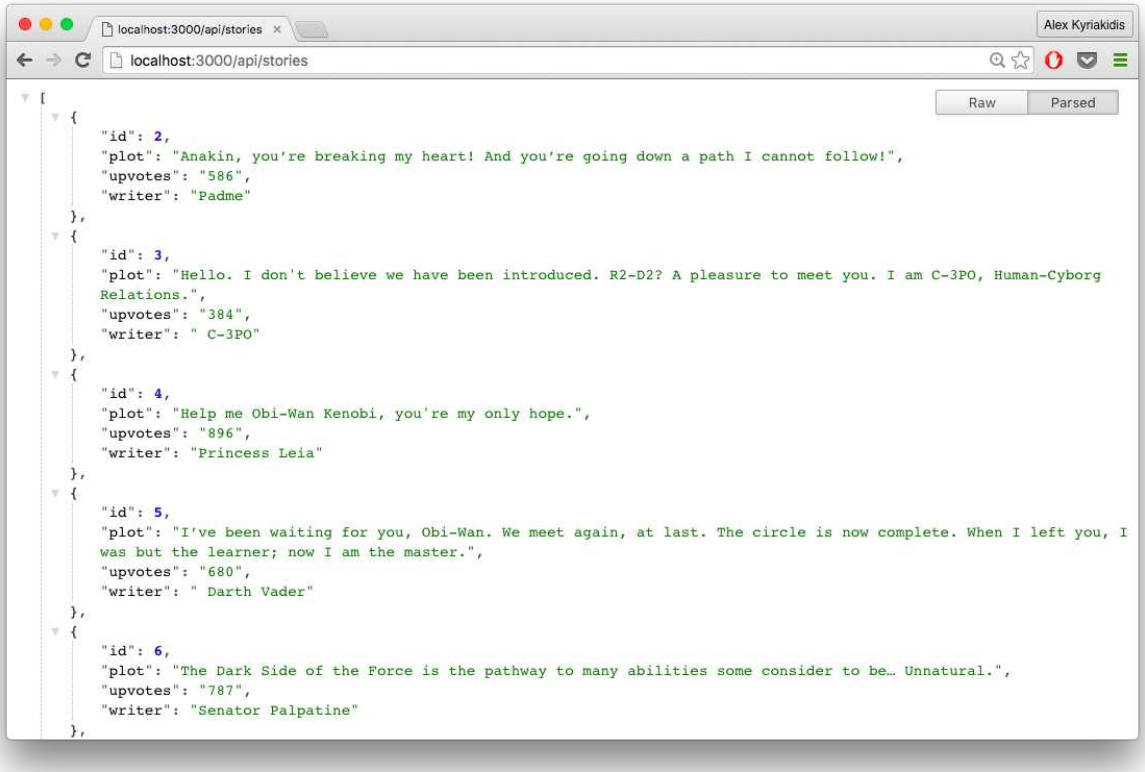
An endpoint is simply a URL. When you go to http://example.com/foo/bar it is an endpoint and you simply need to call it /foo/bar because the domain will be the same for all the endpoints.

To manage the **`Story`** resource we need 5 endpoints. Each endpoint corresponds to a specific action.

HTTP Method	URI	Action
GET/HEAD	api/stories	*Fetches* all stories
GET/HEAD	api/stories/{id}	*Fetches* specified story
POST	api/stories	*Creates* a new story
PUT/PATCH	api/stories/{id}	*Updates* an existing story
DELETE	api/stories/{id}	*Deletes* specified story

As indicated in the above table, to get a listing with all the 'stories' we have to make an HTTP GET or HEAD request to `api/stories`. To update an existing `story` we have to make an HTTP PUT or PATCH request to `api/stories/{storyID}` providing the data we want to override, and replacing `{storyID}` with the id of the story we want to update. The same logic applies to all endpoints. I think you get the idea.

Assuming your server is running on `http://localhost:3000`, you can view a listing of all stories in JSON format by visiting `http://localhost:3000/api/stories` on your web browser.



JSON response

Tip

Reading raw JSON data on browser can be painful. It is always easier to read a **well formatted JSON**. Chrome has some great extensions that could format raw JSON data into tree view format that can be easily read.

I use JSONFormatter[3] because it supports syntax highlighting and displays JSON in tree view, where the nodes on the tree can be collapsed or expanded by clicking the triangle icon on the left of each node. It also provides a button for switching to original (raw) data.

You can choose whichever extension you like but you should **definitely use one!**

[3] https://chrome.google.com/webstore/detail/json-formatter/bcjindcccaagfpapjjmafapmmgkkhgoa

9. Working with real data

It is time to actually put in use our database and perform the operations we have mentioned (CRUD). We will utilize the last example from the Components chapter, but this time, of course, our data will come from an external source. To exchange data with the server we need to perform asynchronous HTTP (Ajax) requests.

Info

AJAX is a technique that allows web pages to be updated asynchronously by exchanging small amounts of data with the server behind the scenes.

9.1 Get Data Asynchronous

Take a moment to have a look at the last example from the Components chapter. As you can see we hardcode **stories array** inside the `data` object of `Vue` instance.

Stories array hardcoded

```
new Vue({
    data: {
        stories: [
            {
                plot: 'My horse is amazing.',
                writer: 'Mr. Weebl',
            },
            {
                plot: 'Narwhals invented Shish Kebab.',
                writer: 'Mr. Weebl',
            },
            ...
        ]
    }
})
```

This time, we want to fetch the existing stories from the server.

To do so , we'll perform a HTTP GET request using jQuery at first. Later on this chapter, we will migrate to vue-resource[1] to see the differences between the two of them.

To make the AJAX call we are going to use **`$.get()`**, a jQuery function that loads data from the server using a HTTP GET request. Full documentation for `$.get()` can be found here[2].

Info

vue-resource is a plugin for `Vue.js` that provides services for making web requests and handle responses.

The **`$.get()`** method's syntax is

```
$.get(
    url,
    success
);
```

which is actually a shorthand for

```
$.ajax({
    url: url,
    success: success
});
```

So what we do now? We want to get the stories from the server using `$.get('/api/stories')` passing the appropriate URL and put the response data we get, inside the `stories` array.

There is a common catch here, we have to make the call **after the document has finished rendering**. Fortunately, there is a helper function called `ready` in `Vue.js` *(similar to `$( document ).ready()`)* which triggers once the page Document Object Model (DOM) is ready.

Lets see this in action.

[1] https://github.com/vuejs/vue-resource

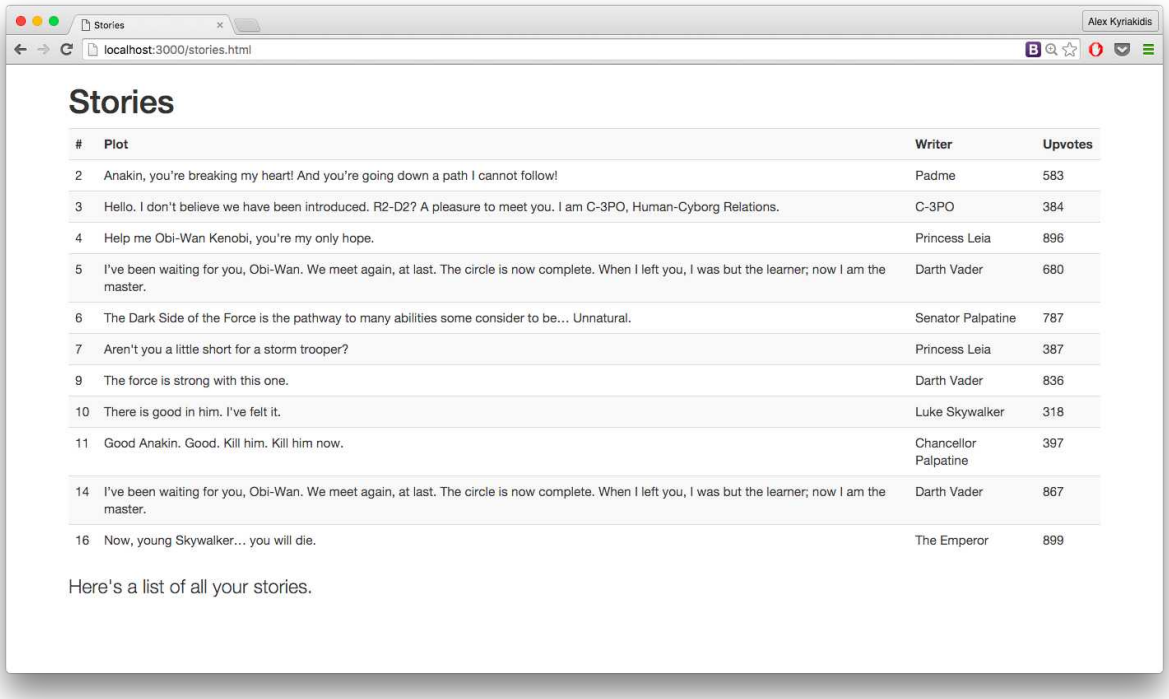
[2] https://api.jquery.com/jquery.get/

```
<div id="app">
    <div class="container">
        <h1>Let's hear some stories!</h1>
        <ul class="list-group">
            <story v-for="story in stories" :story="story">
            </story>
        </ul>
        <pre>{{ $data | json }}</pre>
    </div>
</div>
<template id="template-story-raw">
    <li  class="list-group-item">
        {{ story.writer }} said "{{ story.plot }}"
    <span>{{story.upvotes}}</span>
    </li>
</template>
```

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.js"></script>
<script src="https://code.jquery.com/jquery-2.1.4.min.js"></script>
<script type="text/javascript">
Vue.component('story', {
    template: "#template-story-raw",
    props: ['story'],
});

var vm = new Vue({
    el: '#app',
    data: {
        stories: []
    },
    ready : function(){
        $.get('/api/stories', function(data){
            vm.stories = data;
        })
    }
})
</script>
```

We start by pulling in the jQuery from the cdnjs[3]. Then use the `ready` function and inside it, perform the `GET` request. After the request is successfully finished, we set the response data (inside the callback) to `stories` array.

[3] https://cdnjs.com/libraries/jquery/



Get stories

Notice here, that inside the callback we are referring to `stories` variable using `vm.stories` instead of `this.stories`. We do so because variable **`this`** does not represent the **`Vue`** instance inside the callback. So, we set the whole **`Vue`** instance to a variable called **`vm`**, in order to have access to it from anywhere within our code. To learn more about **`this`**, have a look at the MDN documentation[4].

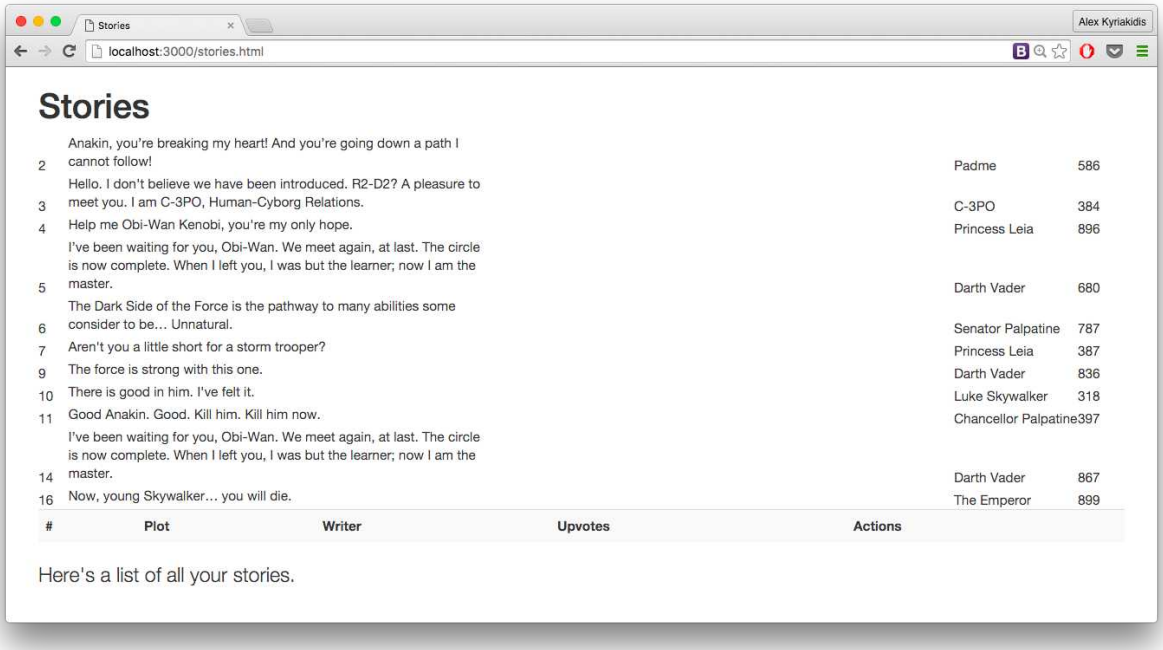
9.2 Refactoring

Having large amounts of code in our text editor can be confusing if not displayed properly, as well as in the browser. For that reason, we are going to refactor our example code, to render the list of `stories` using a **`<table>`** element instead of the **`<ul>`**.

[4] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this

```
<div id="app">
    <table class="table table-striped">
        <tr>
            <th>#</th>
            <th>Plot</th>
            <th>Writer</th>
            <th>Upvotes</th>
            <th>Actions</th>
        </tr>
        <story v-for="story in stories" :story="story">
        </story>
    </table>
    <template id="template-story-raw">
        <tr>
            <td>
                {{story.id}}
            </td>
            <td>
                <span>
                    {{story.plot}}
                </span>
            </td>
            <td>
                <span>
                    {{story.writer}}
                </span>
            </td>
            <td>
                {{story.upvotes}}
            </td>
        </tr>
    </template>
    <p class="lead">Here's a list of all your stories.
    </p>
    <pre>{{ $data | json }}</pre>
</div>
```

But there is an issue.



Rendering issues

Our table does not render properly, but why?[5]. The reason behind this is below:

> *Some HTML elements, for example `<table>`, have restrictions on what elements can appear inside them. Custom elements that are not in the whitelist will be hoisted out and thus not render properly. In such cases you should use the* ***`is`*** *special attribute to indicate a custom element.*

Therefore, to solve this issue we have to use Vue's **special attribute `is`**.

```
<table>
  <tr is="my-component"></tr>
</table>
```

So our example will become

```
<tr v-for="story in stories" is="story" :story="story"></tr>
```

[5] http://goo.gl/Xr9RoQ

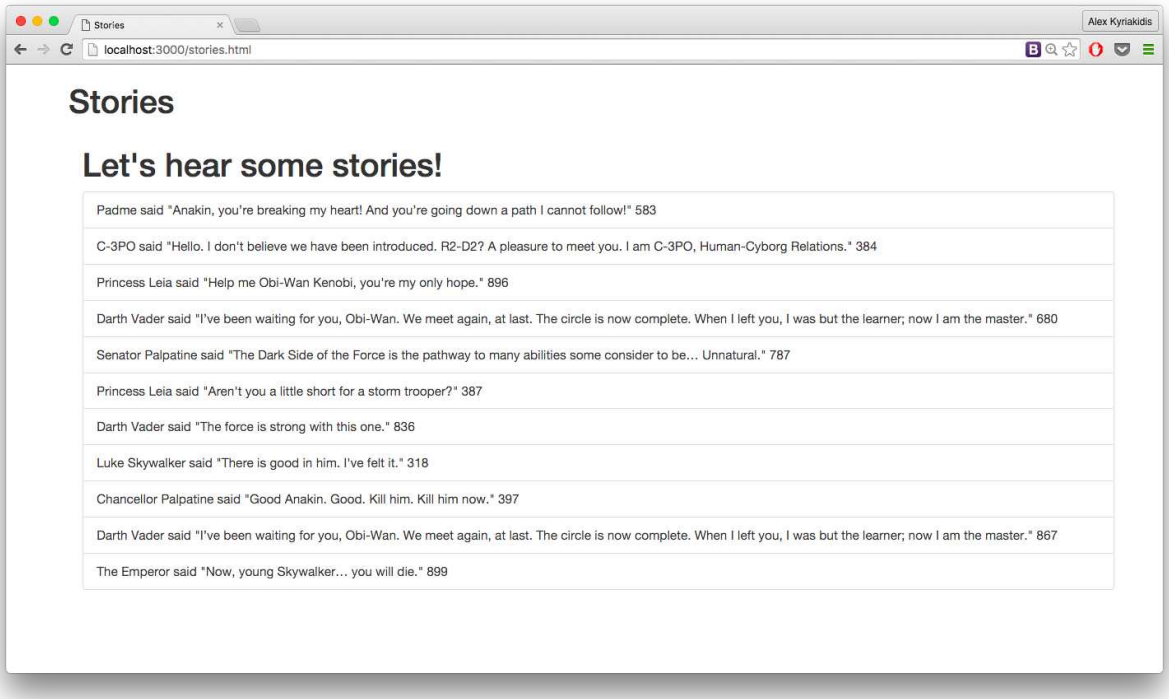


Table renders properly

Well this looks better!

9.3 Update Data

We used to have a function that allowed the user to vote any story he wanted to. But now we want something more. We want the server to be informed every time a story is voted, ensuring that story votes are updated in the database as well.

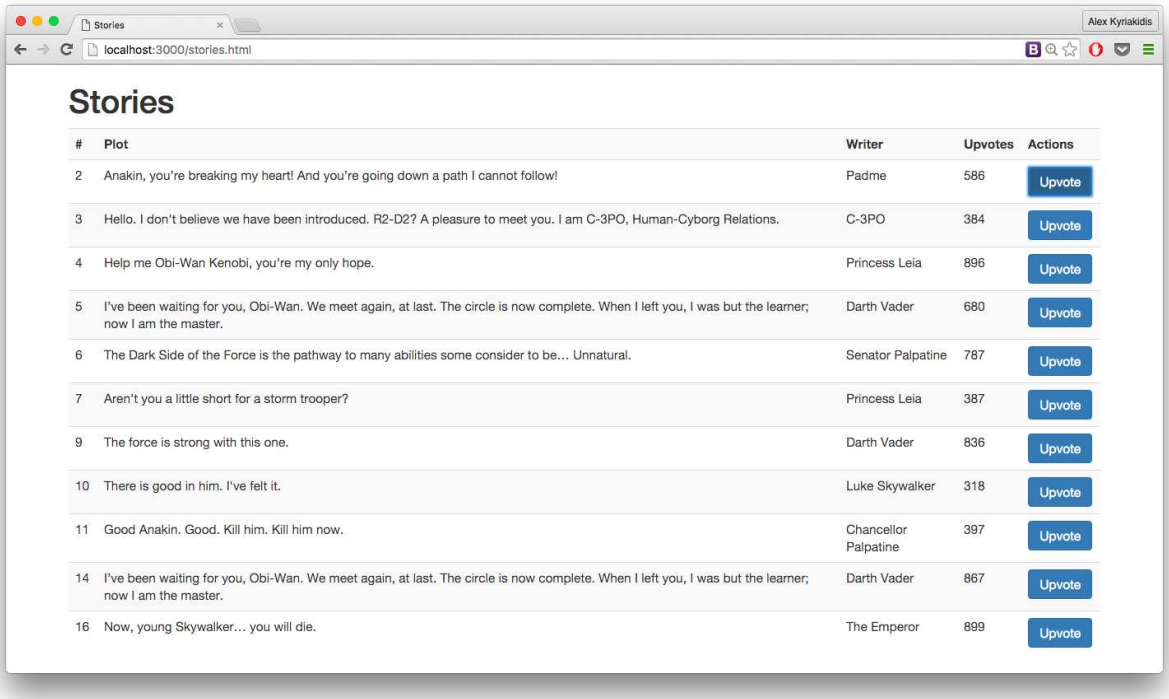
To update an existing story we have to make an HTTP PATCH or PUT request to `api/stories/{storyID}`.

Inside the `upvoteStory` function, which is to be created, we are going to make a HTTP call, after we have increased **`story upvotes`**. We will pass the newly updated **`story`** variable in the Request Payload, in order to update the data in our server.

```
<td>
    <div class="btn-group">
        <button @click="upvoteStory(story)" class="btn btn-primary">
            Upvote
        </button>
    </div>
</td>
```

```
Vue.component('story',{
    template: '#template-story-raw',
    props: ['story'],
    methods: {
        upvoteStory: function(story){
            story.upvotes++;
            $.ajax({
                url: '/api/stories/'+story.id,
                type: 'PATCH',
                data: story,
            });
        }
    },
})
```

We brought back the `upvote` method and placed it inside our `story` component. Making a PATCH request now, providing the new data, the server updates the `upvotes` count.



Upvote stories

9.4 Delete Data

Now let us proceed to another piece of functionality our `stories` list should have: Deleting a story we don't like. To remove a `story` from the array and the DOM, we are going to use Vue's `$remove()` method, which searches for an item and removes it from target Array.

Info

`$remove()` method works as follows. When you want to remove an **`item`** from an array called **`items`** you can do:

```
vm.items.$remove(item)
```

```
<td>
    <div class="btn-group">
        <button @click="upvoteStory(story)" class="btn btn-primary">
            Upvote
        </button>
        <button @click="deleteStory(story)" class="btn btn-danger">
            Delete
        </button>
    </div>
</td>
```

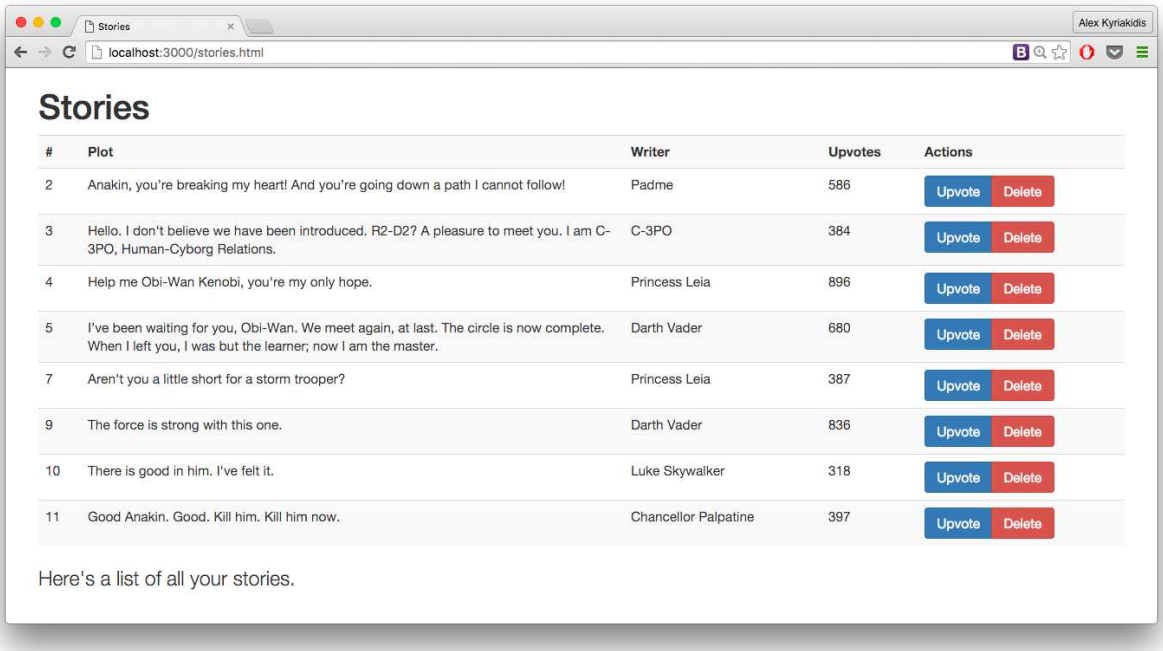
We append a 'Delete' button to the 'actions' column, binded to a method to delete the story. The **`deleteStory`** method will be:

```
Vue.component('story',{
    ...
    methods: {
        ...
        deleteStory: function(story){
            vm.stories.$remove(story)
        }
    }
    ...
})
```

But of course, this way, we will only remove the story temporary. In order to delete the story from the database, we have to perform an AJAX DELETE request.

```
Vue.component('story',{
    ...
    methods: {
        ...
        deleteStory: function(story){
            vm.stories.$remove(story)
            $.ajax({
                url: '/api/stories/'+story.id,
                type: 'DELETE'
            });
        },
    }
    ...
})
```

We are passing in the URL, as we did before. The type here should be equal to `DELETE`. Our method is now ready and we can delete the story from our database as well as the DOM.



Upvote and Delete stories

That's it for now. We will continue our example in the next chapter, by enhancing the functionality with **Creating new stories**, **Editing current stories** and more. But first of all, we will replace **jQuery** with **vue-resource**.

10. Integrating vue-resource

10.1 Overview

Vue-resourse is a resource plugin for Vue.js.

This plugin provides services for making web requests and handles responses using an XMLHttpRequest or JSONP.

We are going to make again all the web requests we made above, using this plugin instead. This way you can see the differences and decide which one is better for your needs. jQuery is nice, but if you are using it only to perform AJAX calls, you may consider removing it.

Here you can find installation instructions and documentation about vue-resource[1] As usual, we are going to "pull it in" from the cdnjs[2] page.

To fetch data from a server and bring them up to display in the browser, we can use vue-resource `$http` method with the following syntax:

```
ready: function() {
      // GET request
      this.$http({url: '/someUrl', method: 'GET'})
      .then(function (response) {
          // success callback
      }, function (response) {
          // error callback
      });
}
```

Info

A Vue instance provides the `this.$http(options)` function which takes an options object for generating an HTTP request and returns a promise. Also the Vue instance will be automatically bound to **`this`** in all function callbacks.

Instead of passing the method option, there are shortcut methods available for all the requested types.

[1] https://github.com/vuejs/vue-resource

[2] https://cdnjs.com/libraries/vue-resource

Request shortcuts

```
this.$http.get(url, [data], [options]).then(successCallback, errorCallback);
this.$http.post(url, [data], [options]).then(successCallback, errorCallback);
this.$http.put(url, [data], [options]).then(successCallback, errorCallback);
this.$http.patch(url, [data], [options]).then(successCallback, errorCallback);
this.$http.delete(url, [data], [options]).then(successCallback, errorCallback);
```

10.2 Migrating

It is time to use `vue-resource` in our example. First of all, we have to include it. We will add this line to our HTML file.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue-resource/0.7.0/vue-resou\
rce.js"></script>
```

To fetch the stories we will make a GET request in the corresponding form.

```
ready: function() {
  // GET request
  this.$http({url: '/api/stories', method: 'GET'}).then(function (response) {
      this.$set('stories', response.data)
      //Or we as we did before
      //vm.stories = response.data
  })
}
```

Our list of stories comes without any problems using the syntax above.

Lets move on now with the DELETE and PATCH requests using the shortcut methods.

PATCH request

```
upvoteStory: function(story){
    story.upvotes++;
    this.$http.patch('/api/stories/'+story.id , story)
},
```

We have replaced the AJAX method with this one, in no time!

DELETE request

```
deleteStory: function(story){
    this.$parent.stories.$remove(story)
    this.$http.delete('/api/stories/'+story.id )
},
```

We observe that the delete function works as expected and the story is deleted from the database.

10.3 Enhancing Functionality

We should add a couple more features to make our list of stories neat. We can give the user the ability to **change the plot and the writer of a story and create new stories**.

10.3.1 Edit Stories

Let's start with the first task and give the user some inputs to manipulate the story's attributes. Two binded inputs should do the job, but we should display them **only** when the user is **editing** a story. It seems like the kind of work we did in previous chapters.

To define if a `story` is in **edit state** we will use a property, **`editing`**, which will become true when the user hits the 'Edit' button.

```
<td>
    <!--if editing story display the input for plot-->
    <input v-if="story.editing" v-model="story.plot" class="form-control">
    </input>
    <!--in other occasions show the story plot-->
    <span v-else>
        {{story.plot}}
    </span>
</td>
<td>
    <!-- if editing story display the input for writer -->
    <input v-if="story.editing" v-model="story.writer" class="form-control">
    </input>
    <!--in other occasions show the story writer-->
    <span v-else>
        {{story.writer}}
    </span>
</td>
```

```
<td>
    {{story.upvotes}}
</td>
<td>
    <div v-if="!story.editing" class="btn-group">
        <button @click="upvoteStory(story)" class="btn btn-primary">
            Upvote
        </button>
        <button @click="editStory(story)" class="btn btn-default">
            Edit
        </button>
        <button @click="deleteStory(story)" class="btn btn-danger">
            Delete
        </button>
    </div>
</td>
```

```
Vue.component('story',{
    ...
    methods: {
        ...
        editStory: function(story){
            story.editing=true;
        },
    }
    ...
})
```

This is our updated table with two new inputs and a button. We use the `editStory` function to set `story.editing` to `true` , so `v-if` will bring up the inputs to edit the story and hide the 'Upvote' and 'Delete' buttons. But this approach won't work. It seems that the DOM isn't updating after setting `story.editing` to `true`. But why does this happen?

It turns out, according to this post from Vue.js blog[3] that when **you are adding a new property that wasn't present when the data was observed**, the DOM won't update. The best practice is to always declare properties that need to be reactive **upfront**. In cases where you absolutely need to add or delete properties at runtime, use the global `Vue.set` or `Vue.delete` methods.

For this reason, we have to initialize the `story.editing` attribute to `false` on each story, **right after** receiving the stories from the server.

To do this, we are going to use javascript's `.map()` method within the success callback of the GET request.

[3] http://vuejs.org/2016/02/06/common-gotchas/

```
ready: function() {
    // GET request
    this.$http({url: '/api/stories', method: 'GET'}).then(function (response) {
        var storiesReady = response.data.map(function(story){
            story.editing = false;
            return story
        })

        this.$set('stories', storiesReady)
    })
}
```

Info

The `.map()` method calls a defined callback function on each element of an array and returns an array that contains the results. You can find more information about the `.map()` method and its syntax Here[4]

The function passes the new attribute inside each story object and then returns the updated story. The new variable `storiesReady` is an array that contains our updated array with the new attribute on.

When the story is under edit, we will give the user two options: to update the story with new values and to cancel the edit.

#	Plot	Writer	Upvotes	Actions
12	Laugh it up, Fuzz ball.	Han Solo	279	Upvote Edit Delete
14	Fear is the path to the dark side.	Yoda	561	
15	I find your lack of faith disturbing.	Darth Vader	582	Upvote Edit Delete
16	Laugh it up, Fuzz ball.	Han Solo	772	Upvote Edit Delete

Form inputs for story editing

So, lets move on and add two new buttons that should be displayed only when the user is editing a story. Additionally a new method called `updateStory` will be created, which updates the current editing story, after the 'Update Story' button is pressed.

[4] https://msdn.microsoft.com/en-us/library/ff679976(v=vs.94).aspx

```
<!-- If story is under edit, display this group of buttons -->
<div class="btn-group" v-else>
    <button @click="updateStory(story)" class="btn btn-primary">
        Update Story
    </button>
    <button @click="story.editing=false" class="btn btn-default">
        Cancel
    </button>
</div>
```

```
Vue.component('story',{
    ...
    methods: {
        ...
        updateStory: function(story){
            this.$http.patch('/api/stories/'+story.id , story)
            //Set editing to false to show actions again and hide the inputs
            story.editing = false;
        },
    }
    ...
})
```

Updating story actions

Here it is up and running. After the PATCH request is finished successfully, we have to set `story.editing` to `false` in order to hide the inputs and bring back the action buttons.

10.3.2 Create New Stories

Now for a bit trickier task, we are going to give the user the ability to create a new story and save it to our server. First, we must provide inputs so the new story can be typed in. To make this happen, we will create an empty story and we'll add it to the stories array using the **`push()`** javascript method. We will initialize all the story's attributes to null, except from **`editing`**. We want to immediately manipulate the story, so the **`editing`** will be set to **`true`**.

```
var vm = new Vue({
    ...
    methods: {
        createStory: function(){
            var newStory={
                "plot": "",
                "upvotes": 0,
                "editing": true
            };
            this.stories.push(newStory);
        },
    }
})
```

```
<p class="lead">Here's a list of all your stories.
    <button @click="createStory()" class="btn btn-primary">
        Add a new one?
    </button>
</p>
```

Info

The push() method adds new items to the end of an array, and returns the new length. You can find more information about the `push()` method and its syntax Here[5]

As soon as the new variable is set, we push it to our stories array. We named the new function **`createStory`** and we placed it in our Vue instance.

Right bellow our list, we have added a button. When the button is clicked, **`createStory`** method gets invoked. Since the `newStory.editing` is set to true, the binded inputs for "plot" and "writer" along with the 'Edit action buttons', are being rendered instantly.

Also, the new story object must be sent to the server in order to be stored in the database. We are going to perform a POST request inside a method called `storeStory`.

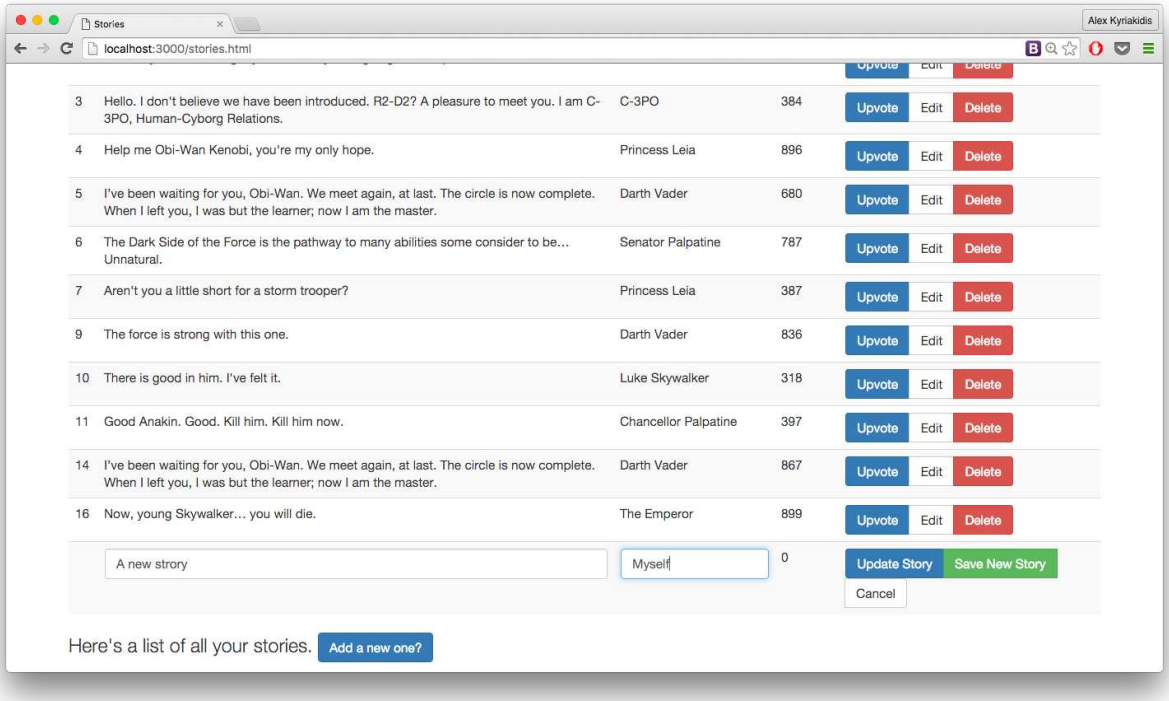
[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/push

```
Vue.component('story',{
    ...
    methods: {
        ...
        storeStory: function(story){
            this.$http.post('/api/stories/', story).then(function() {
                story.editing = false;
            });
        },
    }
    ...
})
```

We 've used the shortcut method for the request type POST. Also, in the success callback function we have set editing to false in order to show the 'action buttons' again and hide the form's inputs and 'editing' buttons. Below we update the button "groups", in accordance to the new method.

```
<td>
<div class="btn-group" v-if="!story.editing">
    <button @click="upvoteStory(story)" class="btn btn-primary">
        Upvote
    </button>
    <button @click="editStory(story)" class="btn btn-default">
        Edit
    </button>
    <button @click="deleteStory(story)" class="btn btn-danger">
        Delete
    </button>
</div>
<div class="btn-group" v-else>
    <button  class="btn btn-primary" @click="updateStory(story)">
        Update Story
    </button>
    <button  class="btn btn-success" @click="storeStory(story)">
        Save New Story
    </button>
    <button @click="story.editing=false" class="btn btn-default">
        Cancel
    </button>
</div>
</td>
```

We observe a small mistake in this block of code. When we are in "editing" mode (`v-else` block) we see that the buttons for update and store are being shown together, but we only need one for each story, since each story will be `Stored` **or** `Updated`. It can't do both. So, if the story is an old one and the user is about to edit it, we need the update button. On the other hand, if the story is new, we need the store button.



A small mistake

To bypass this issue, we are going to restructure our buttons. The `Update` button will **only** be displayed when the **story is old**. Accordingly the `Save new` button will be displayed when the story **is a new one**.

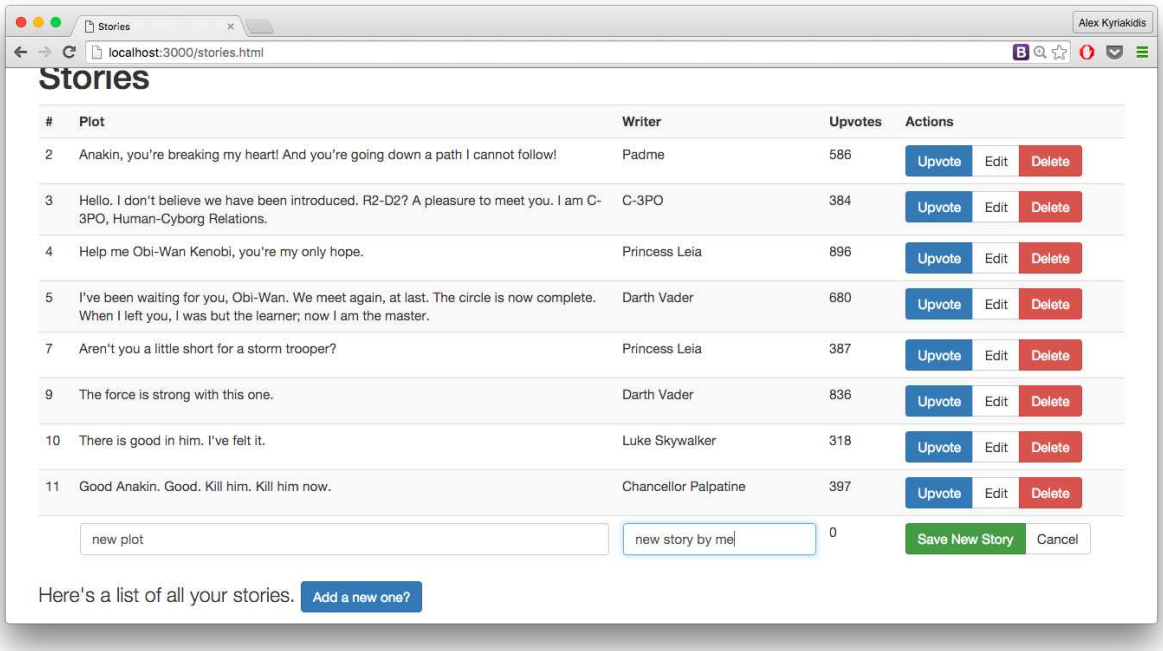
You may have noticed that all stories fetched from the server have an `id` attribute. We are going to use this observation to **define if a story is new or not**.

```
<div class="btn-group" v-else>
    <!--If the story is an old one then we want to update it
    TIP: if the story is taken from the db then it will have an id-->
    <button v-if="story.id" class="btn btn-primary" @click="updateStory(story)">
        Update Story
    </button>
    <!--If the story is new we want to store it-->
    <button v-else class="btn btn-success" @click="storeStory(story)">
        Save New Story
```

```
10      </button>
11      <!--always show cancel-->
12      <button @click="story.editing=false" class="btn btn-default">
13          Cancel
14      </button>
15  </div>
```


Tip

If the story is taken from the database then it will have an id.



Adding new story

So there we have it. It wasn't that hard, right?

After finishing this part, testing our app shows another error. After creating, saving, and trying to edit a new story, we see that the button says "Save new Story" instead of "Update Story"! Thats because we are not fetching the newly created story from the server, after we send it, and it does not have an id yet. To solve this problem we can fetch the stories from server again, after finishing creating a story.

Since I don't like to repeat my code, I will extract the fetching procedure to a method called **`fetchStories()`**. After that, I can use this method to fetch the stories anytime.

The fetchStories method

```
var vm = new Vue({
    el: '#v-app',
    data : {
        stories: [],
    },

    ready : function(){
        this.fetchStories()
    },
    methods: {
        createStory: function(){
            var newStory={
                "plot": "",
                "upvotes": 0,
                "editing": true

            };
            this.stories.push(newStory);
        },
        fetchStories: function () {
            this.$http.get('/api/stories')
                .then(function (response) {
                    var storiesReady = response.data.map(function(story){
                        story.editing = false;
                        return story
                    })
                    this.$set('stories', storiesReady)
                });
        },
    }
});
```

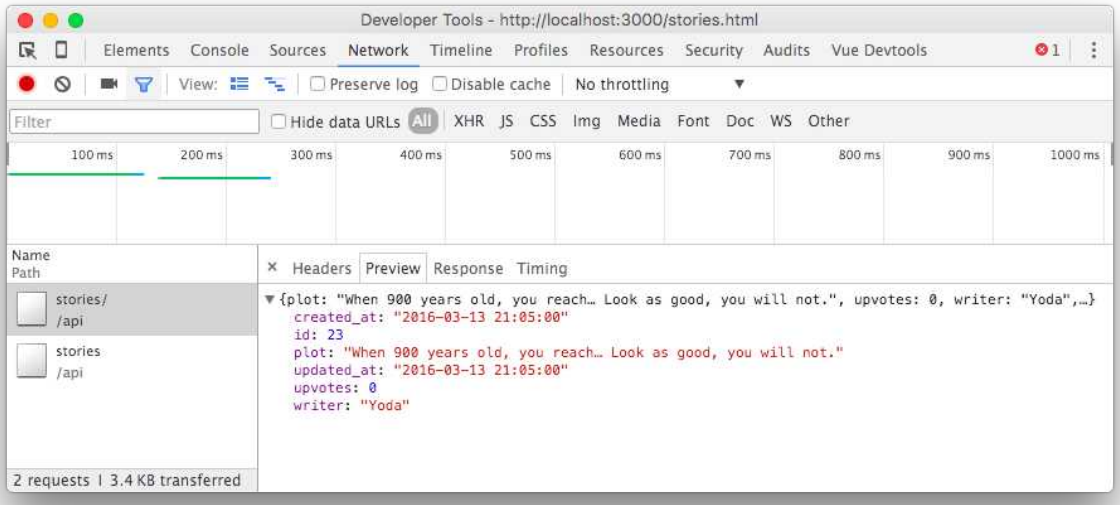
In our situation, we'll call `fetchStories()` inside the success callback of the POST request.

```
Vue.component('story',{
    ...
    methods: {
        ...
        storeStory: function(story){
            this.$http.post('/api/stories/', story).then(function() {
                story.editing = false;
                vm.fetchStories();
            });
        },
    }
    ...
})
```

That's it! We can now create and edit any story we want.

10.3.3 Store & Update Unit

A better way to fix the previous issue, is to update only the newly created story from the database, instead of fetching and overwriting all the stories. If you see the server response for the POST request you will see that it returns the created story along with its id.



Server response after creating new story

The only thing we have to do, is to update our story to match the server's one. So, we will set the `id` of the response data, to story's id attribute. We will do this inside the POST's success callback.

```
Vue.component('story',{
    ...
    methods: {
        ...
        storeStory: function(story){
            this.$http.post('/api/stories/', story).then(function(response) {
                Vue.set(story, 'id', response.data.id);
                story.editing = false;
            });
        },
    }
    ...
})
```

I use **`Vue.set(story, 'id', response.data.id)`** instead of **`story.id = response.data.id`** because inside our table we display the **id** of each story. Since the new story had no **`id`**, when it is pushed to the `stories` array, the DOM won't be updated when the **`id`** changes, so we will not be able to see the new **`id`**.

Tip

When you are adding a **new property that wasn't present when the data was observed**, Vue.js cannot detect the property's addition. So, if you need to add or remove properties at runtime, use the global **`Vue.set`** or **`Vue.delete`** methods.

10.4 JavaScript File

As you may have noticed, our code is becoming big, and as our project grows, it will be hard to maintain. For starters, we'll separate the JavaScript code from the HTML. I'll create a file called **`app.js`** and I'll save it under **`js`** directory.

All the JavaScript code should live inside that file from now on. To include the newly created script to any HTML page you simply have to add this tag

```
<script src='/js/app.js' type="text/javascript"></script>
```

and you are **ready to go!**

10.5 Source Code

Below is the whole source code of the previous *Managing Stories* example. Because the code is big enough I suggest you to open your local files with your favorite text editor, if you have downloaded our repo. The file is located at `~/themajestyofvuejs/apis/stories/`.

If you haven't downloaded the repository you can still view the **stories.html**[6] and **app.js**[7] files on *github*.

stories.html

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Stories</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2\
/css/bootstrap.min.css">
</head>
<body>
<main>
    <div class="container">
        <h1>Stories</h1>
        <div id="v-app">
            <table class="table table-striped">
                <tr>
                    <th>#</th>
                    <th>Plot</th>
                    <th>Writer</th>
                    <th>Upvotes</th>
                    <th>Actions</th>
                </tr>
                <tr v-for="story in stories" is="story" :story="story"></tr>
            </table>
            <template id="template-story-raw">
                <tr>
                    <td>
                        {{story.id}}
                    </td>
                    <td class="col-md-6">
```

[6] https://github.com/hootlex/the-majesty-of-vuejs/blob/master/apis/stories/public/stories.html
[7] https://github.com/hootlex/the-majesty-of-vuejs/blob/master/apis/stories/public/js/app.js

```
31                          <input v-if="story.editing" v-model="story.plot"
32                                     class="form-control">
33                          </input>
34                          <!--in other occasions show the story plot-->
35                          <span v-else>
36                              {{story.plot}}
37                          </span>
38                      </td>
39                      <td>
40                          <input v-if="story.editing" v-model="story.writer"
41                                   class="form-control">
42                          </input>
43                          <!--in other occasions show the story writer-->
44                          <span v-else>
45                              {{story.writer}}
46                          </span>
47                      </td>
48                      <td>
49                          {{story.upvotes}}
50                      </td>
51                      <td>
52                          <div class="btn-group" v-if="!story.editing">
53                              <button @click="upvoteStory(story)"
54                                  class="btn btn-primary"
55                              >
56                                  Upvote
57                              </button>
58                              <button @click="editStory(story)"
59                                  class="btn btn-default"
60                              >
61                                  Edit
62                              </button>
63                              <button @click="deleteStory(story)"
64                                  class="btn btn-danger"
65                              >
66                                  Delete
67                              </button>
68                          </div>
69                          <div class="btn-group" v-else>
70                              <!--If the story is taken from the db then it will h\
71  ave an id-->
72                              <button v-if="story.id"
```

```
                                @click="updateStory(story)"
                                class="btn btn-primary"
                            >
                                Update Story
                            </button>
                            <!--If the story is new we want to store it-->
                            <button v-else
                                @click="storeStory(story)"
                                class="btn btn-success"
                            >
                                Save New Story
                            </button>
                            <!--Always show cancel-->
                            <button @click="story.editing=false"
                                class="btn btn-default"
                            >
                                Cancel
                            </button>
                        </div>
                    </td>
                </tr>
            </template>
            <p class="lead">Here's a list of all your stories.
                <button @click="createStory()"
                    class="btn btn-primary"
                >
                    Add a new one?
                </button>
            </p>
            <pre>{{ $data | json }}</pre>
        </div>
    </div>
</main>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.min.js"></scr\
ipt>
<script src="https://code.jquery.com/jquery-2.1.4.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue-resource/0.7.0/vue-resou\
rce.js"></script>
<script src='/js/app.js' type="text/javascript"></script>
</body>
</html>
```

```
Vue.component('story', {
    template: '#template-story-raw',
    props: ['story'],
    methods: {
        deleteStory: function (story) {
            this.$parent.stories.$remove(story)
            this.$http.delete('/api/stories/' + story.id)
        },
        upvoteStory: function (story) {
            story.upvotes++;
            this.$http.patch('/api/stories/' + story.id, story)
        },
        editStory: function (story) {
            story.editing = true;
        },
        updateStory: function (story) {
            this.$http.patch('/api/stories/' + story.id, story)
            // Set editing to false to show actions again and hide the inputs
            story.editing = false;
        },
        storeStory: function (story) {
            this.$http.post('/api/stories/', story)
            .then(function (response) {
                // After the the new story is stored in the database
                // we fetch again all stories
                // vm.fetchStories();
                // OR Better, update the id of the created story
                Vue.set(story, 'id', response.data.id);
                // Set editing to false to show actions again and hide the inputs
                story.editing = false;
            });
        },
    }
})
new Vue({
    el: '#v-app',
    data: {
        stories: [],
    },
    ready: function () {
        this.fetchStories()
    },
```

```
    methods: {
        createStory: function () {
            var newStory = {
                plot: "",
                upvotes: 0,
                editing: true
            };
            this.stories.push(newStory);
        },
        fetchStories: function () {
            var vm = this;
            this.$http.get('/api/stories')
                .then(function(response){
                    // set data on vm
                    var storiesReady = response.data.map(function (story) {
                        story.editing = false;
                        return story
                    })
                    vm.$set('stories', storiesReady)
                });
        },
    }
});
```

Code Examples

You can find the code examples of this chapter on GitHub[8]

10.6 Homework

To get comfortable with making web requests and handling responses you should replicate what we did in this chapter.

What you have to do is to consume an API in order to:

- create a table and **display existing** movies
- **modify** existing movies
- **store** new movies in the database
- **delete** movies from the database

I have prepared **the database and the API** for you. You only have to write HTML and JavaScript.

10.6.1 Preface

If you have followed the instructions from Chapter 8 open your terminal and run:

```
cd ~/themajestyofvuejs/apis/movies
sh setup.sh
```

If you haven't, you should run this:

```
mkdir ~/themajestyofvuejs
cd ~/themajestyofvuejs
git clone https://github.com/hootlex/the-majesty-of-vuejs .
cd ~/themajestyofvuejs/apis/movies
sh setup.sh
```

You now have a **database filled with great movies** along with a fully functional server **running on 'http://localhost:3000'!**

To ensure that everything is working fine browse to 'http://localhost:3000/api/movies' and you should see an array of movies in JSON format.

10.6.2 API Endpoints

The **API Endpoints** you are going to need are:

[8] https://github.com/hootlex/the-majesty-of-vuejs/tree/master/examples/10.%20Integrating%20vue-resource

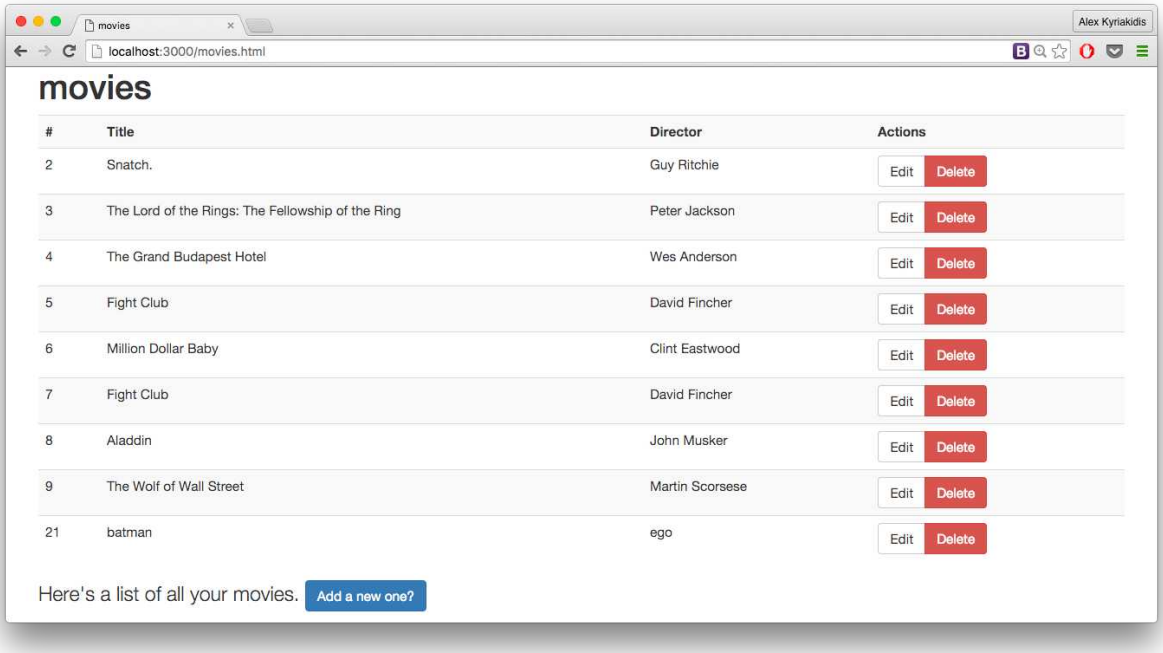
HTTP Method	URI	Action
GET/HEAD	api/movies	*Fetches* all movies
GET/HEAD	api/movies/{id}	*Fetches* specified movie
POST	api/movies	*Creates* a new movie
PUT/PATCH	api/movies/{id}	*Updates* an existing movie
DELETE	api/movies/{id}	*Deletes* specified movie

10.6.3 Your Code

Put your HTML code inside ~/themajestyofvuejs/apis/movies/public/movies.html file we have created. You can place your JavaScript code there too, or inside js/app.js.

To check your work visit '**http://localhost:3000/movies.html**' with your browser.

I hope you will enjoy this one, Good luck!



Example Output

You can find a potential solution to this exercise here[9].

[9] https://github.com/hootlex/the-majesty-of-vuejs/blob/master/homework/chapter10

11. Pagination

In the previous chapter we managed to fetch all the records from the database and display them inside a table. That implementation is fine for a bunch of records, but in the real world, when you have to work with thousands or millions of records, you cannot simply fetch them and place them inside an `array`. If you do so, your browser will not be happy to load such an amount of data, but even if it manages to do that, then I assure you that no user likes dealing with a table containing 100,000 rows.

Info

Pagination is used in some form in almost every web application to divide returned data and display it on multiple pages. Pagination also includes the logic of preparing and displaying the links to the various pages, and it can be handled client-side or server-side. Server-side pagination is more common.

In situations like this, the developers who designed the API will (hopefully) divide the returned data in pages.

The HTTP response will contain some simple **`meta-data`** next to the main data, to inform you about *total* items, *per page* items, etc. To help you traverse through the pages, it will provide information such as **`the current page`**, **`the next page`**, and **`the previous page`**.

Example Response with Paginated data

```
{
    "total": 10000,
    "per_page": 50,
    "current_page": 15,
    "last_page": 200,
    "next_page_url": "/api/stories?page=16",
    "prev_page_url": "/api/stories?page=14",
    "from": 751,
    "to": 800,
    "data": [...]
}
```

The pagination's `meta-data` could also be inside an object next to **`data`**, or anywhere the API developers have decided.

Example Response with Paginated data

```
{
    "data": [...],
    "pagination": {
        "total": 10000,
        "per_page": 50,
        "current_page": 15,
        "last_page": 200,
        "next_page_url": "/api/stories?page=16",
        "prev_page_url": "/api/stories?page=14",
        "from": 751,
        "to": 800,
    }
}
```

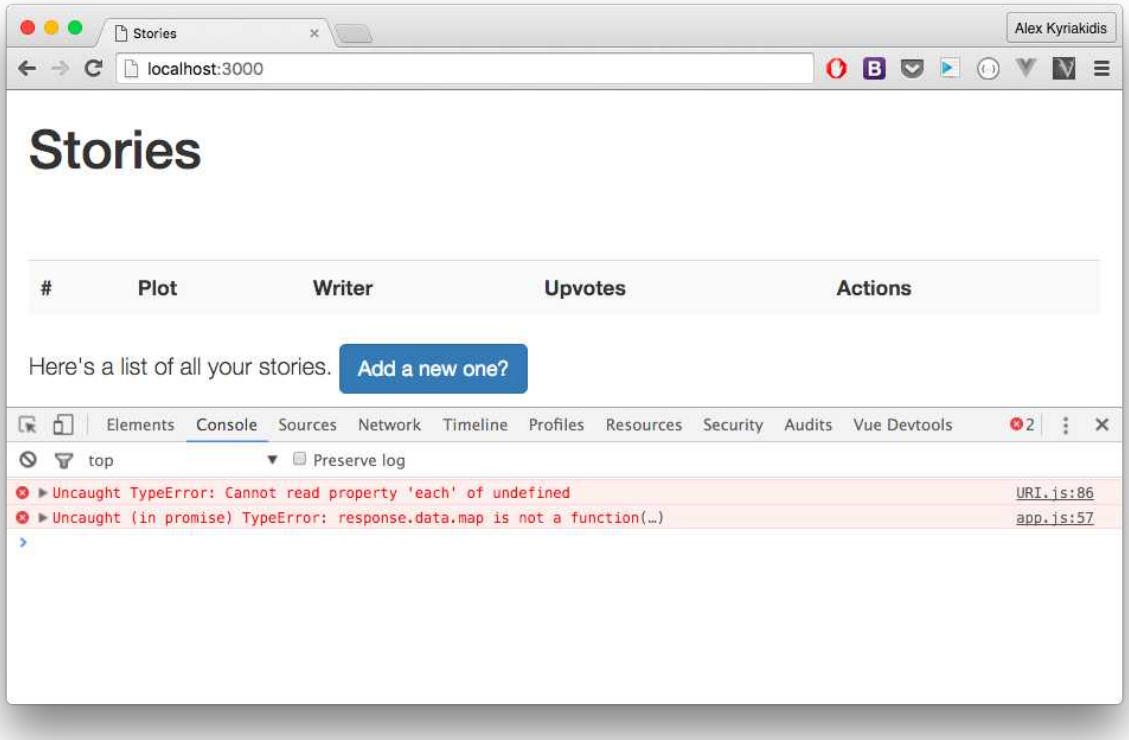
11.1 Implementation

We are going to continue working with the *story* examples from the previous chapter, using the slightly improved paginated API. So, we are going to modify the code, to be able to access and use the data.

If you take a look at the code from the previous example you will see that our **fetchStories** method is similar to this:

```
new Vue({
    ...
    methods: {
        ...
        fetchStories: function () {
            var vm = this;
            this.$http.get('/api/stories')
                .then(function (response) {
                    var storiesReady = response.data.map(function (story) {
                        story.editing = false;
                        return story
                    })
                    vm.$set('stories', storiesReady)
                });
        },
        ...
    }
});
```

If we open our HTML file on the browser, as you may have already guessed, our table doesn't render properly.

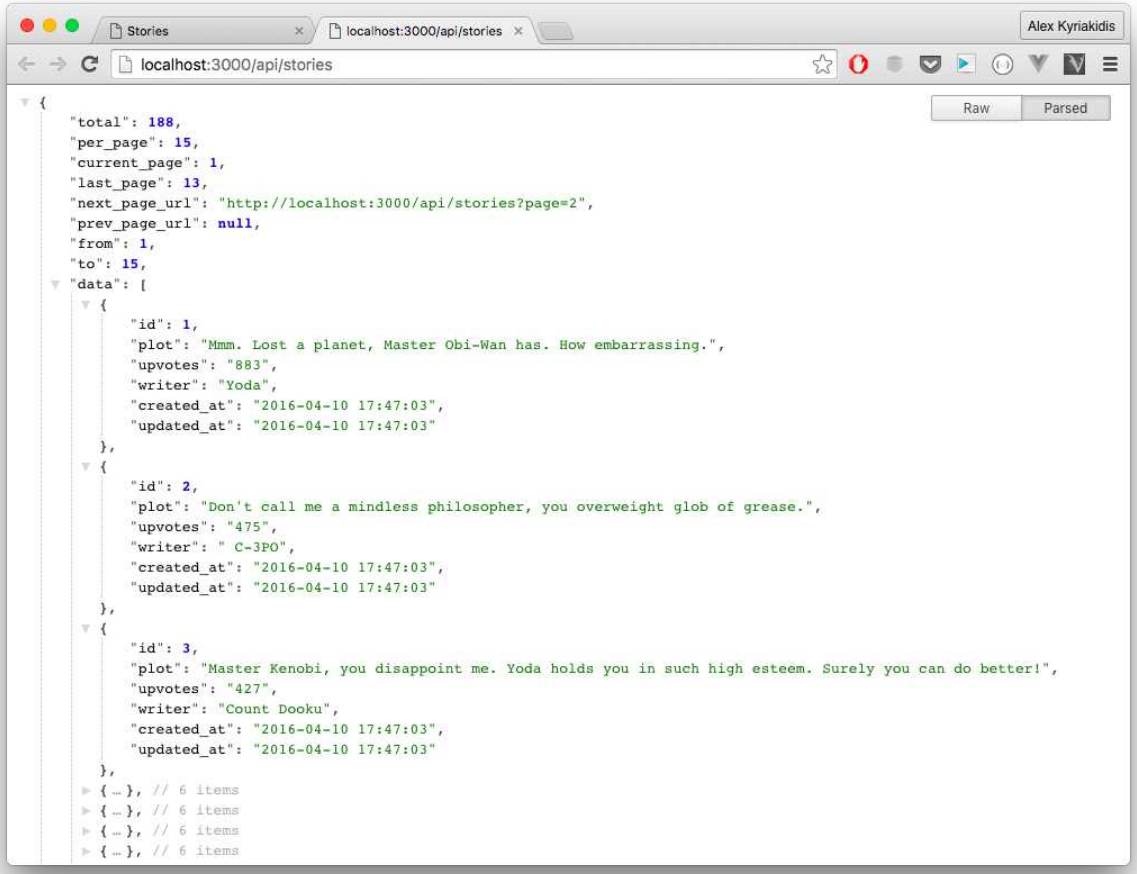


Stories' aren't displaying

This happens because the `stories` are now returned inside `data` array. To fix this, we have to change `response.data` to `response.data.data` (I know this is kinda weird, but…).

Now, except from the stories array, we also want to save the pagination's data inside an object in order to easily implement the pagination functionality.

To find out how we can access those data, let's have a look at the server's response.



Server's response

For starters, we don't need all those data. So, we will stick with *current_page*, *last_page*, *next_page_url*, and *prev_page_url*.

Our pagination object will be something like this:

```
pagination: {
    "current_page": 15,
    "last_page": 200,
    "next_page_url": "/api/stories?page=16",
    "prev_page_url": "/api/stories?page=14"
}
```

Let's modify our **fetchStories** method to update **pagination** object each time it fetches stories from the database.

```
new Vue({
    ...
    methods: {
        ...
        fetchStories: function () {
            var vm = this;
            this.$http.get('/api/stories')
            .then(function (response) {
                var storiesReady = response.data.data.map(function (story) {
                    story.editing = false;
                    return story
                })
                //here we use response.data
                var pagination = {
                    current_page: response.data.current_page,
                    last_page: response.data.last_page,
                    next_page_url: response.data.next_page_url,
                    prev_page_url: response.data.prev_page_url
                }
                vm.$set('stories', storiesReady)
                vm.$set('pagination', pagination)
            });
        },
        ...
    }
});
```

11.2 Pagination Links

By now, we have our **`pagination`** object but we always fetch the first page of `stories` since we are making a GET HTTP request to 'api/stories'. We have to change the requested page based on user input (next page, previous page).

First we'll update the **`fetchStories`** method to accept an argument with the desired page. If no argument is passed, it will fetch the first page. I'll also create a new method **`makePagination`** to make the code cleaner.

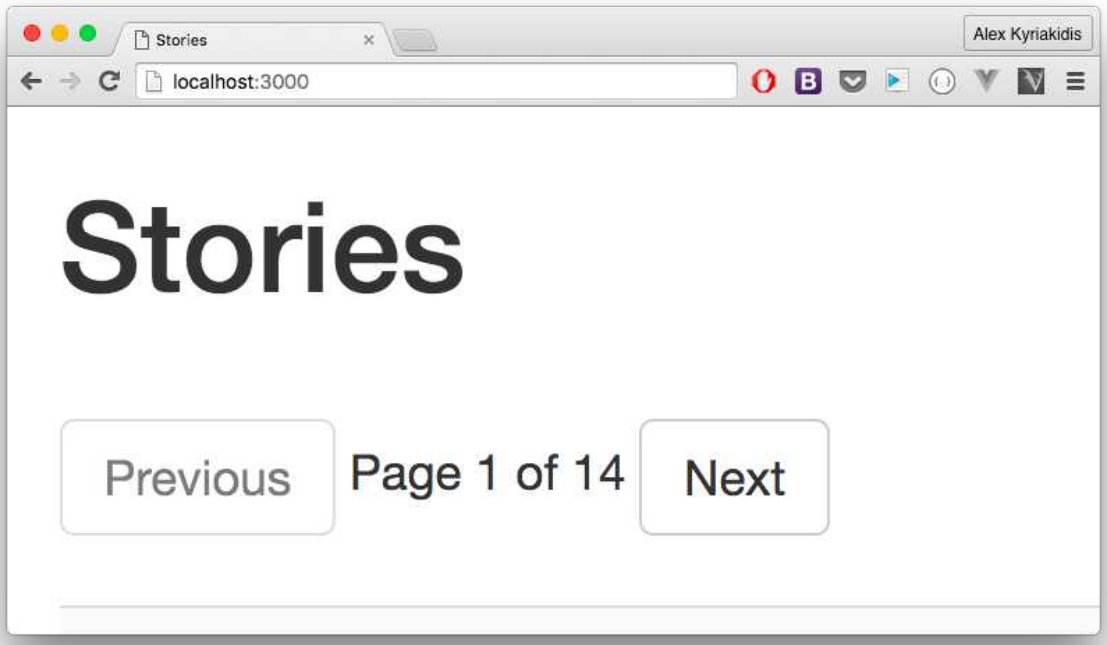
```
new Vue({
    ...
    methods: {
        ...
        fetchStories: function (page_url) {
            var vm = this;
            page_url = page_url || '/api/stories'
            this.$http.get(page_url)
                .then(function (response) {
                    var storiesReady = response.data.data.map(function (story) {
                        story.editing = false;
                        return story
                    })
                    vm.makePagination(response.data)
                    vm.$set('stories', storiesReady)
                });
        },
        makePagination: function (data){
            //here we use response.data
            var pagination = {
                current_page: data.current_page,
                last_page: data.last_page,
                next_page_url: data.next_page_url,
                prev_page_url: data.prev_page_url
            }
            this.$set('pagination', pagination)
        }
        ...
    }
}
```

Now that our method is ready, we need a way to call it properly. We will add 2 buttons, one for next and one for previous page, on the top of our **`#app div`**. Each button will call **`fetchStories`** method when clicked, passing the corresponding **page url**.

```
<div class="pagination">
    <button @click="fetchStories(pagination.prev_page_url)">
        Previous
    </button>
    <button @click="fetchStories(pagination.next_page_url)">
        Next
    </button>
 </div>
```

Pomp! If you try to click the buttons you'll see that they work as expected. We've got our pagination in the blink of an eye. It will be useful though to inform the user which page he is currently looking at, and the total number of pages. Also, we can disable the **previous** button when the user is on the first page, and the **next** on the last accordingly.

```
<div class="pagination">
    <button @click="fetchStories(pagination.prev_page_url)"
            :disabled="!pagination.prev_page_url"
    >
        Previous
    </button>
    <span>Page {{pagination.current_page}} of {{pagination.last_page}}</span>
    <button @click="fetchStories(pagination.next_page_url)"
            :disabled="!pagination.next_page_url"
    >
        Next
    </button>
 </div>
```



Disabled previous button

Code Examples

You can find the code examples of this chapter on GitHub[1].

11.3 Homework

There is nothing particular to do for homework in this chapter. If you actually want to work on this example, I will provide you the paginated API.

If you have solved the previous chapter's homework (downloaded the code and started a server), you are just a few clicks away. If you haven't, just follow these instructions.

The paginated API lives inside *'~/themajestyofvuejs/apis/pagination/stories'* directory.

The HTML file is in *'~/themajestyofvuejs/apis/pagination/stories/public'* directory.

If you just want to view the final code you can take a look at the files on GitHub[2].

[1] https://github.com/hootlex/the-majesty-of-vuejs/tree/master/examples/11.%20Pagination
[2] https://github.com/hootlex/the-majesty-of-vuejs/tree/master/apis/pagination/stories/public

Building Large-Scale Applications

12. ECMAScript 6

Before we take things a step further and see how we can build Large-Scale Applications, I would like to familiarize you with ECMAScript 6.

Info

ECMAScript is a client-side scripting language's specification, that is the basis of several programming languages including JavaScript, ActionScript, and JScript.

ECMAScript 6 (ES6), also known as ES2015, is the latest version of the ECMAScript standard. The ES6 specification was finalized in June 2015. It is a significant update to the language, and the first major update since ES5 was standardized in 2009. Implementation of ES6 features in major JavaScript engines is underway now[1].

12.1 ES6 Features

ES6 has a lot of new features. We are going to review those that we will use in the next chapters. If you are interested in learning more about what is new in ES6, I highly recommend you the book "Understanding ECMAScript 6" by Nicholas C. Zakas which is available on leanpub[2]. There is also an online version[3] of the book for free.

Also, there are other useful resources and tutorials like the one on Babel[4], an article on tutsplus[5], a blog post[6] by Nicholas C. Zakas again, and a ton of stuff around the web!

[1] http://kangax.github.io/compat-table/es6/
[2] https://leanpub.com/understandinges6/
[3] https://leanpub.com/understandinges6/read
[4] https://babeljs.io/docs/learn-es2015/
[5] http://code.tutsplus.com/articles/use-ecmascript-6-today--net-31582
[6] https://www.nczonline.net/blog/2013/09/10/understanding-ecmascript-6-arrow-functions/

12.1.1 Compatibility

Unsuprisingly, support varies wildly from engine to engine, with Mozilla tending to lead the way. ES6 compatibility table[7] is a useful start for establishing what ECMAScript 6 features your browser does and doesn't support.

Note

If you're using Chrome most of the ES6 features are hidden behind a feature toggle. Browse to chrome://flags, find the section titled "Enable Experimental JavaScript" and enable it to turn on support

From now on we will develop our examples using ES6 features.

12.1.2 Variable Declarations

12.1.2.1 Let Declarations

`let` is the new **`var`**. You can basically replace var with let to declare a variable, but limit the variable's scope only to the current code block. Since let declarations are not hoisted to the top of the enclosing block, you better always place let declarations first in the block, so that they are available to the entire block. For example:

Let inside if

```
let age = 22
if (age >= 18) {
    let adult = true;
    console.log(adult); //outputs true
}
//adult isn't accessible here
console.log(adult);
//ERROR: Uncaught ReferenceError: adult is not defined
```

[7] https://kangax.github.io/compat-table/es6/

Let on top

```
let age = 22
let adult
if (age >= 18) {
    adult = true;
    console.log(adult); //outputs true
}
//now adult is accessible here
console.log(adult); //outputs true
```

12.1.2.2 Constant Declarations

Constants, like let declarations, are block-level declarations. There is one big difference between let and const. Once you declare a variable using const, it is defined as a constant, which means that **you can't change its value**.

```
const name = "Alex"

name = "Kostas" //throws error
```

Info

Much like constants in other languages, their value cannot change later on. However, unlike constants in other languages, the value a constant holds **may be modified if it is an object**.

12.1.3 Arrow Functions

One of the most interesting new parts of ECMAScript 6 is the arrow functions. Arrow functions are functions defined with a new syntax that uses an "arrow" (⇒). They support both expression and statement bodies. Unlike functions, arrows share the same lexical `this` as their surrounding code.

For example, the following arrow function takes a single argument and returns its value increased by 1:

```
var increment = value => value + 1;
increment(5) //returns 6

// equivalent to:

var increment = function(value) {
    return value + 1;
};
```

Another example with an arrow function which takes 2 arguments and returns their sum:

```
var sum = (a, b) => a + b;
sum(5, 10) //returns 15

// equivalent to:

var sum = function(a, b) {
    return a + b;
};
```

An example arrow function which takes no arguments and uses a statement.

```
var sayHiAndBye = () => {
    console.log('Hi!');
    console.log('Bye!');
};
sayHiAndBye()

// equivalent to:

var sayHiAndBye = function() {
    console.log('Hi!');
    console.log('Bye!');
};
```

12.1.4 Modules

This is, to me, the biggest improvement of the language. ES6 now supports exporting and importing modules across different files.

The simplest example is to create a `.js` file with a variable, and use it inside another file like this:

module.js

```
export name = 'Alex'
```

main.js

```
import {name} from './module'
console.log('Hello', name)
//outputs "Hello Alex"
```

You can also export variables along with functions **one by one**

module.js

```
export var name = 'Alex'
export function getAge(){
  return 22;
}
```

main.js

```
import {name, getAge} from './module'
console.log(name, 'is', getAge())
```

Or **inside an object**:

module.js

```
var name = 'Alex'
function getAge(){
  return 22;
}
export default {name, age}
```

main.js

```
import person from './module'
console.log( person.name, 'is', person.getAge() )
//outputs "Alex is 22"
```

12.1.5 Classes

JavaScript classes are introduced in ECMAScript 6 and are syntactical sugar over JavaScript's existing prototype-based inheritance. The class syntax is **not** introducing a new object-oriented inheritance model to JavaScript. JavaScript classes provide a much simpler and clearer syntax to create objects and deal with inheritance.

Class Example

```
//parent class
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }

  calcArea() {
    return this.height * this.width;
  }

  //To create a getter, use the keyword get.
  get area() {
    return this.calcArea();
  }
  //To create a setter, you do the same, using the keyword set.
}

//child class
class Square extends Rectangle{
  constructor(side) {
    //call parent's constructor
    super(side, side)
  }
}

var square = new Square(5);

console.log(square.area); //outputs 25
```

12.1.6 Default Parameter Values

With ES6 you can define default parameter values.

```
function divide(x, y=2){
    return x/y;
}

// equivalent to:

function divide(x, y){
    y = y == undefined ? 2 : y;
    return x/y;
}
```

13. Advanced Workflow

All these ES6 features (and many more) may get you excited, but there is a catch here. As we mentioned before, **not** all browsers fully support ES6/ES2015 features.

In order to be able to write this new JavaScript syntax today, we need to have a middleman which will take our code and transpile it to Vanilla JS[1], which *every browser understands*. This procedure is really **important** in production, even though you might not think so.

Let me tell you a story. A few years ago, a co-worker of mine began using some cool JS features that weren't fully supported by all browsers. A few days later our users started complaining about some pages of our website not showing correctly, but we couldn't figure out why. We tested it on different PCs, Android phones, iPhones, etc, and it was 100% functional in all our browsers. Later, he found out that older versions of mobile Safari didn't support his code. *Don't be that guy!*

As you can see, some times it's **really hard** to know if the code you write is going to work well on ALL browsers, including Facebook's mobile browser, which is my worst fear.

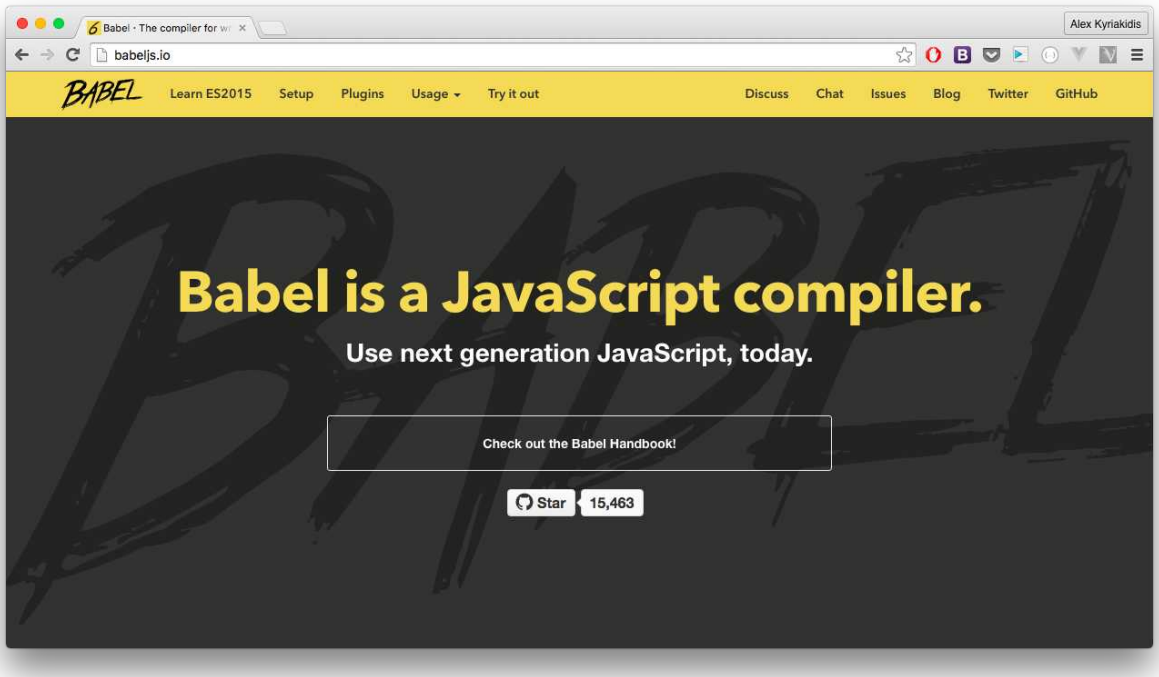
13.1 Compiling ES6 with Babel

Babel will be our Middleman. Babel is a source-to-source JavaScript compiler, which lets us use next generation JavaScript, today.

Info

A source-to-source compiler, transcompiler or transpiler, is a type of compiler that takes the source code of a program written in one programming language, as its input, and produces the equivalent source code in another programming language.

[1] http://vanilla-js.com/



Babel

Before installing Babel, you have to install **Node.js**. To do so, head to Node's website[2] and hit the download button for the the Latest Stable Version. It is going to give you a '.pkg' file (or .msi if you are on Windows). When the download is finished, open the file and follow the instructions. Then, do the required restart, and you are done!

[2] https://nodejs.org/en/

Node.js

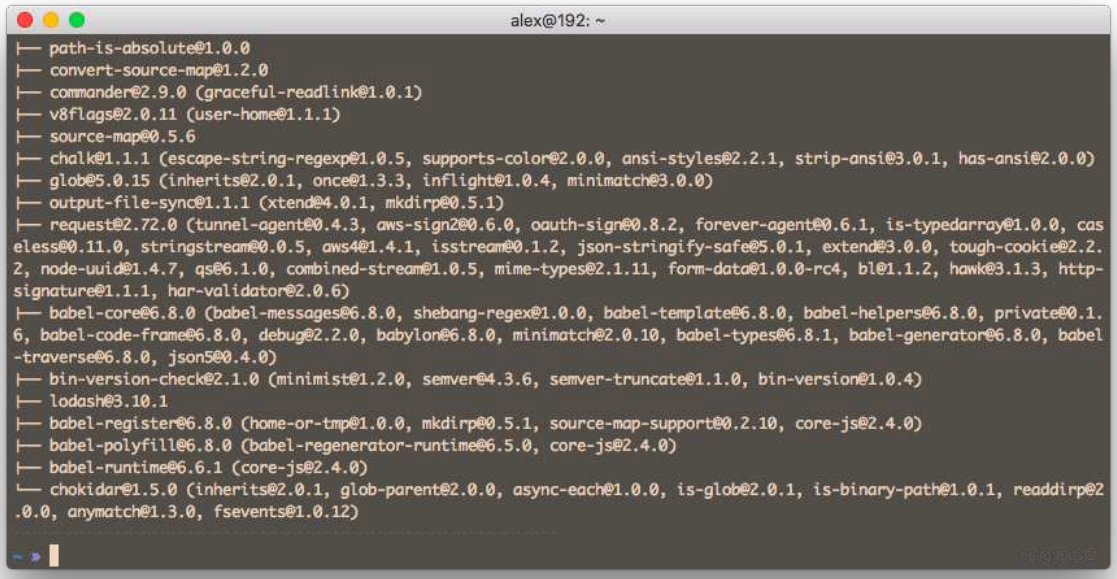
13.1.1 Installation

Create a new directory and place a file named `package.json` inside, containing an empty JSON object (`{}`). You can do it manually, or by running the following commands in your terminal.

```
mkdir babel-example

echo {} > package.json
```

Then run this to install Babel:

```
npm install --save-dev babel-cli
```



Terminal output

When it's done, your package.json file should be something like this:

package.js

```
1  {
2    "devDependencies": {
3      "babel-cli": "^6.8.0"
4    }
5  }
```

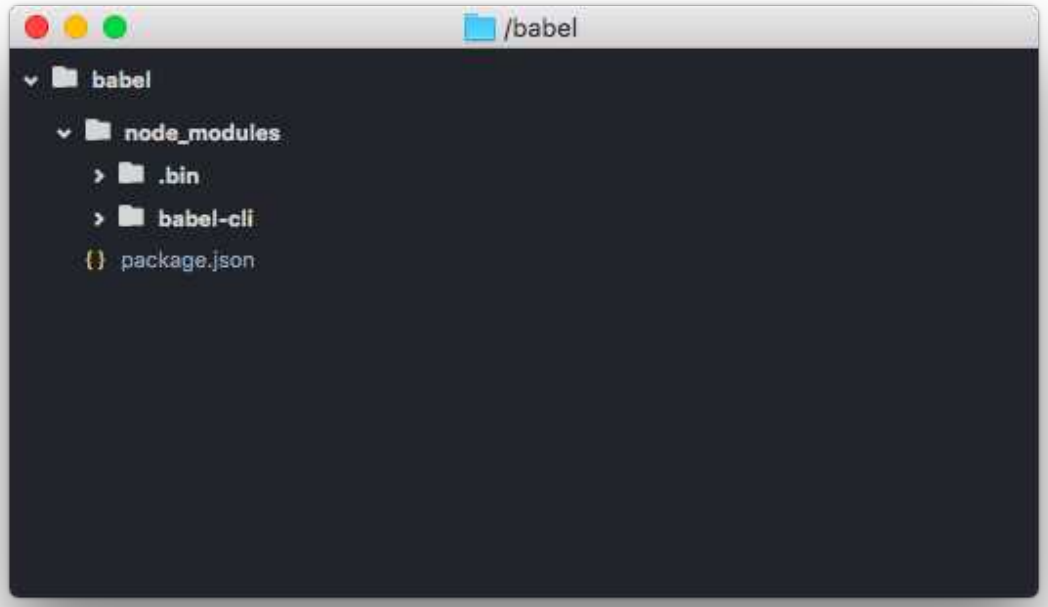
What is `package.json`?

A package.json file contains meta data about your app or module. Most importantly, it includes the list of dependencies to install from npm when running **`npm install`**. If you're familiar with Composer, it's similar to the `composer.json` file.

To learn more about `package.json` have a look at npm docs[3].

Your project's directory should look like this:

[3] https://docs.npmjs.com/files/package.json



Project directory

13.1.2 Configuration

Now that we have babel installed, we need to explicitly tell it what transformations to run on build. Since we want to transform ES2015 code, we will install the ES2015-Preset[4].

We'll also create a config file (`.babelrc`) to enable our preset.

```
npm install babel-preset-es2015 --save-dev
echo { "presets": ["es2015"] } > .babelrc
```

Tip

If the second command fails, enclose file contents inside quotes like this: **`echo '{ "presets": ["es2015"] }' > .babelrc`**

[4] https://babeljs.io/docs/plugins/preset-es2015/

13.1.3 Build alias

Instead of running Babel directly from the command line, we're going to put our commands within **npm scripts**.

We'll add a **`scripts`** field to our **`package.json`** file, and register babel command there, as **`build`**.

Our **`package.json`** will look like this:

package.js

```
{
    "scripts": {
        "build": "babel src -d assets/js"
    },
    "devDependencies": {
        "babel-cli": "^6.8.0",
        "babel-preset-es2015": "^6.6.0"
    }
}
```

This works like an alias. Meaning that when we run **`npm run build`** we are actually running **`babel src -d assets/js`**. This command tells Babel to transpile the code from the **`src`** directory to **`assets/js`** directory.

Before we run the `build` command we have to do a few more things. For starters, go on and create the above-mentioned dirs (*src* and *assets/js*).

13.1.4 Usage

Lets move on and put some files inside *src* folder. I will create a file with a simple `sum` function and call it `sum.js`.

src/sum.js

```
const sum = (a, b) => a + b;
console.log(sum(5,3));
```

Thats was it. We can now run:

```
>_  npm run build
```

When you run it, you can see in your terminal that the **`src\sum.js`** file has been compiled to **`assets\js\sum.js`** and looks like this:

assets/js/sum.js

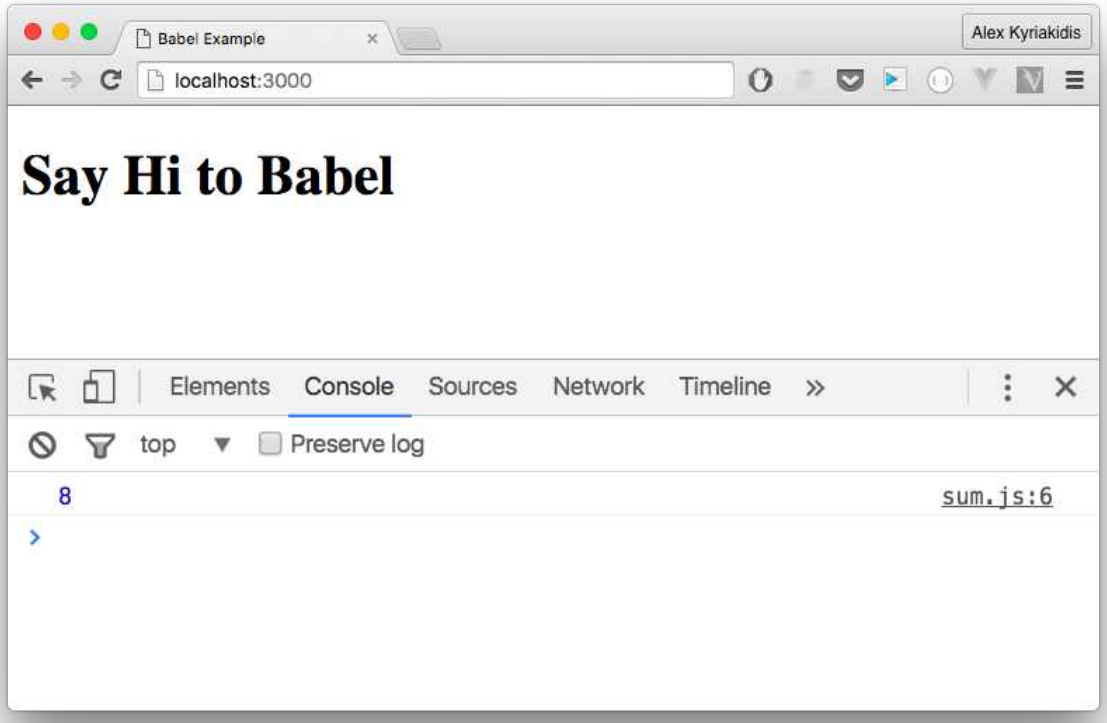
```
"use strict";

var sum = function sum(a, b) {
    return a + b;
};
console.log(sum(5, 3));
```

From now on, whenever you want to compile your ES6 code you can do it by running the **`build`** command. **Pretty neat, huh?**

It's time to see the resulted **`sum.js`** file in the browser. I will create a **`sum.html`** and include our `js`.

sum.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Babel Example</title>
</head>
<body>
    <h1>Babel Example</h1>

    <script src="assets/js/sum.js"></script>
</body>
</html>
```



Browser output

As you can see, the result of the `sum` function is successfully printed to the console.

Info

When you want to test your '.js' file, but you don't want to get in the process of serving it to the browser, you can run it with Node.js.

In the 'sum.js' example there is a 'console.log(sum(5,3))' line, so if you type in your terminal `node sum.js` you 'll see the result (**8**) pop right up!

13.1.5 Homework

This homework exercise aims to help you remember what you've learned by reproducing the example we've built. Instead of the `sum.js` go on and use *ES6 Classes* to create a "Ninja.js" file which will contain a *Ninja* class.

Ninja class should have a property **name** and a method **announce**, which will alert the presence of a Ninja.

For example

```
new Ninja('Leonardo').announce()
//returns "Ninja Leonardo is here!"
```

Don't forget to compile your *js* using Babel before including it in your *HTML*.

Hint

You can find an example for building classes on the previous chapter.

Hint 2

Don't forget to run '**npm run build**' each time you make a change in your *js* file, or it won't update!

You can find a potential solution to this exercise here[5].

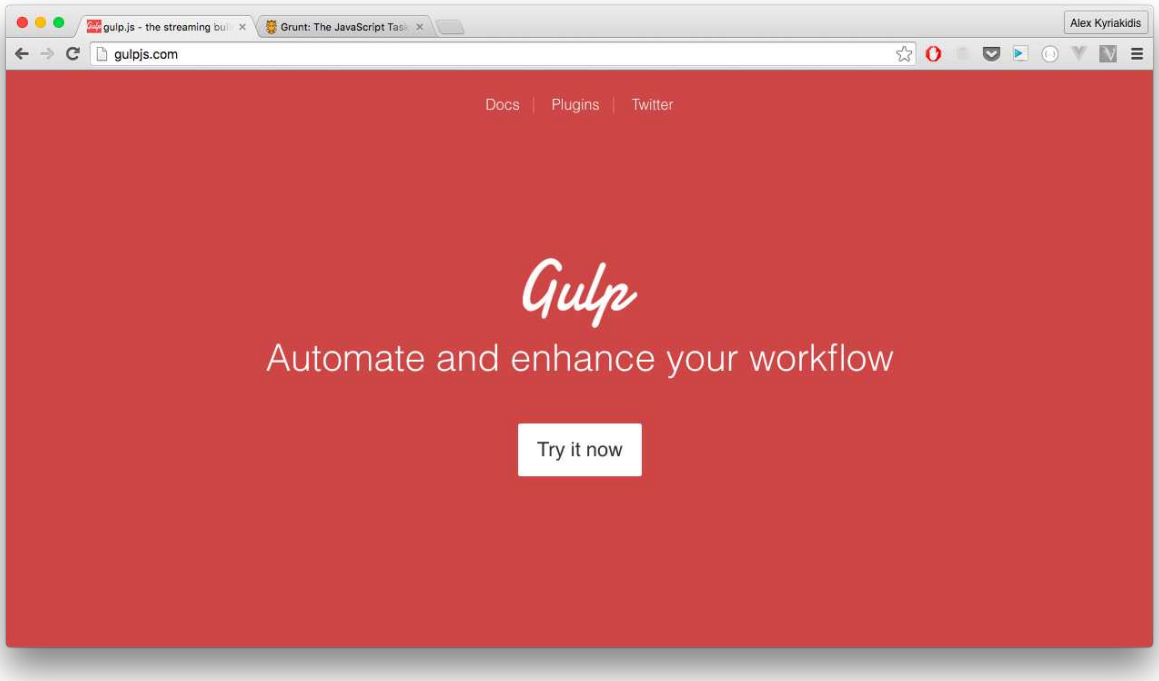
[5] https://github.com/hootlex/the-majesty-of-vuejs/tree/master/homework/chapter13/chapter13.1

13.2 Workflow Automation with Gulp

13.2.1 Task Runners

If you devoted some time to develop the app from the previous Homework section, you have probably found out that it is kinda annoying having to run `npm run build` every time you make a change in your code.

This is where **Task Runners** like Gulp[6] or Grunt[7] come in handy. Task runners let you automate and enhance your workflow.



Gulp

Why use a task runner?

In one word: **Automation**. The less work you have to do when performing repetitive tasks like minification, compilation, unit testing, linting, etc, the easier your job becomes.

[6] http://gulpjs.com/
[7] http://gruntjs.com/

Gulp vs Grunt

Grunt, like Gulp, is a tool for defining and running tasks. The major difference between Grunt and Gulp is that Grunt defines tasks using **configuration objects** while Gulp defines tasks as **JavaScript functions**. Since Gulp runs Javascript, it provides more flexibility in writing your tasks.

Both have a massive plugin library. There, you can find one which does whatever you want, without having to write any code.

13.2.2 Installation

I will show you an example of how you can use Gulp to watch for changes in your `js` files and automatically run the **build** command.

First we have to install Gulp:

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

Then we'll install Gulp in our project's (*babel-example*) devDependencies:

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

Now that we have gulp installed we will create a **`gulpfile.js`** at the root of our project:

gulpfile.js

```
1  const gulp = require('gulp');
2
3  gulp.task('default', function() {
4    // place code for your default task here
5  });
```

13.2.3 Usage

When we now run **`gulp`** in our console, it starts, but does nothing yet. We have to setup a default task.

In order to run `babel` directly, I'll install an npm plugin called gulp-babel[8].

[8] https://www.npmjs.com/package/gulp-babel

```
>_  npm install --save-dev gulp-babel
```

I'll add a new *gulp task* named 'babel' and set it as the default task. My `gulpfile` will look like this:

gulpfile.js

```
const gulp = require('gulp');
const babel = require('gulp-babel');

gulp.task('default', ['babel']);

//basic babel task
gulp.task('babel', function() {
  return gulp.src('src/*.js')
  .pipe(babel({
      presets: ['es2015']
  }))
  .pipe(gulp.dest('assets/js/'))
})
```

This task basically tells babel to transform all `js` files under 'src' directory using the `es2015` preset and put them inside 'assets/js' directory.

13.2.4 Watch

Currently running **`gulp`** on your console has the same effect with **`npm run build`**. What we want to achieve here is to run this task every time a `js` file has changed. To do so, we will set up a *Watcher* inside our `gulpfile` like this:

gulpfile.js

```
const gulp = require('gulp');
const babel = require('gulp-babel');

gulp.task('default', ['watch']);

//basic babel task
gulp.task('babel', function() {
  return gulp.src('src/*.js')
  .pipe(babel({
      presets: ['es2015']
  }))
```

```
12    .pipe(gulp.dest('assets/js/'))
13  })
14
15  //the watch task
16  gulp.task('watch', function() {
17    gulp.watch('src/*.js', ['babel']);
18  })
```

When we run **`gulp watch`** on our console, gulp is watching for changes in our *.js* files under the specified directory. Every time we make a change, gulp runs our *babel* task and our assets/js are being updated. **How awesome is that?**

13.2.5 Homework

This homework exercise is following the previous one. If you haven't done the previous one, it's never too late to begin!

Since this part of the chapter is dedicated to Task Runners, you have to to setup a watcher with Gulp and compile your code with *Babel*, when a change is detected.

Note

You may already have noticed that when running Gulp it prints messages in terminal ("Starting"- "Finished"), so don't be so hasty and wait for the changes to be applied.

You can find a potential solution to this exercise here[9].

[9] https://github.com/hootlex/the-majesty-of-vuejs/tree/master/homework/chapter13/chapter13.2

Gulp is watching!

13.3 Module Bundling with Webpack

13.3.1 Module Bundlers

Our workflow works fine with current code of `sum.js`. We will extend its features to calculate the cost of a pizza and a beer and let the client know.

src/sum.js

```
1  const pizza = 10
2  const beer = 5
3
4  const sum = (a, b) => a + b + '$';
5  console.log('Alex, you have to pay', sum(pizza, beer))
```

This code looks good, but assuming that not everyone is called *Alex* we'll create a new file, **`client.js`**, which will provide the client's name.

src/client.js

```
export const name = 'Alex'
```

We'll import the name from there.

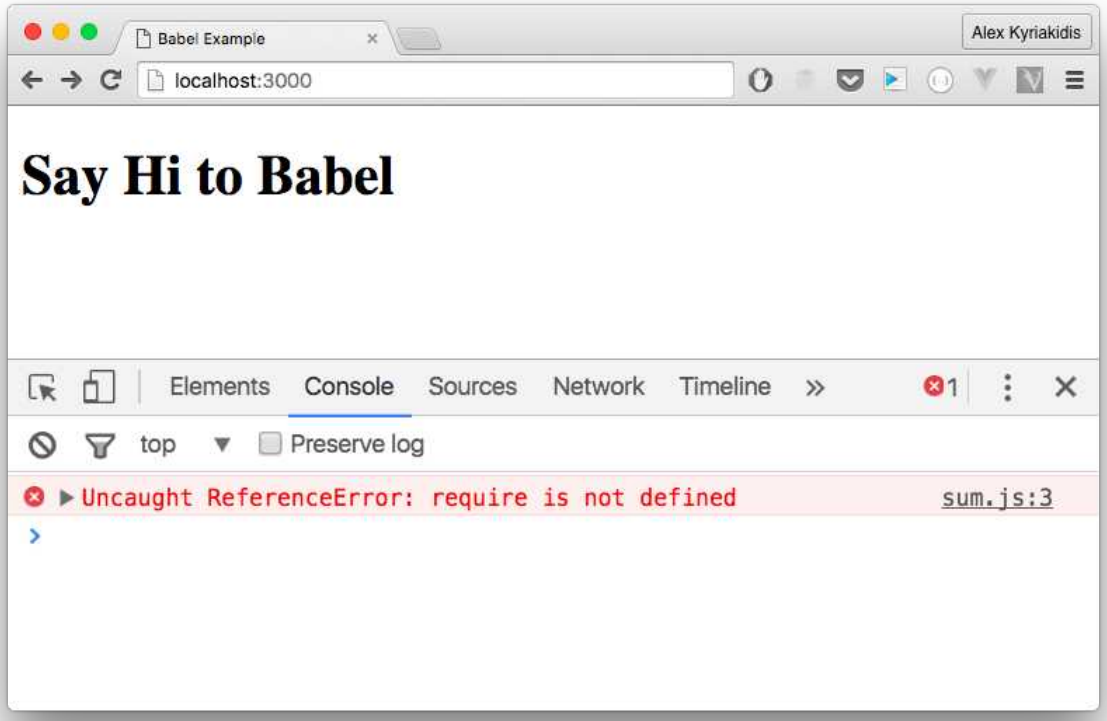
src/sum.js

```
import { name } from './client'

const pizza = 10
const beer = 5

const sum = (a, b) => a + b + '$';
console.log(name, 'you have to pay', sum(pizza, beer))
```

Great! If we run **`node assets/js/sum.js`** we get the expected output.

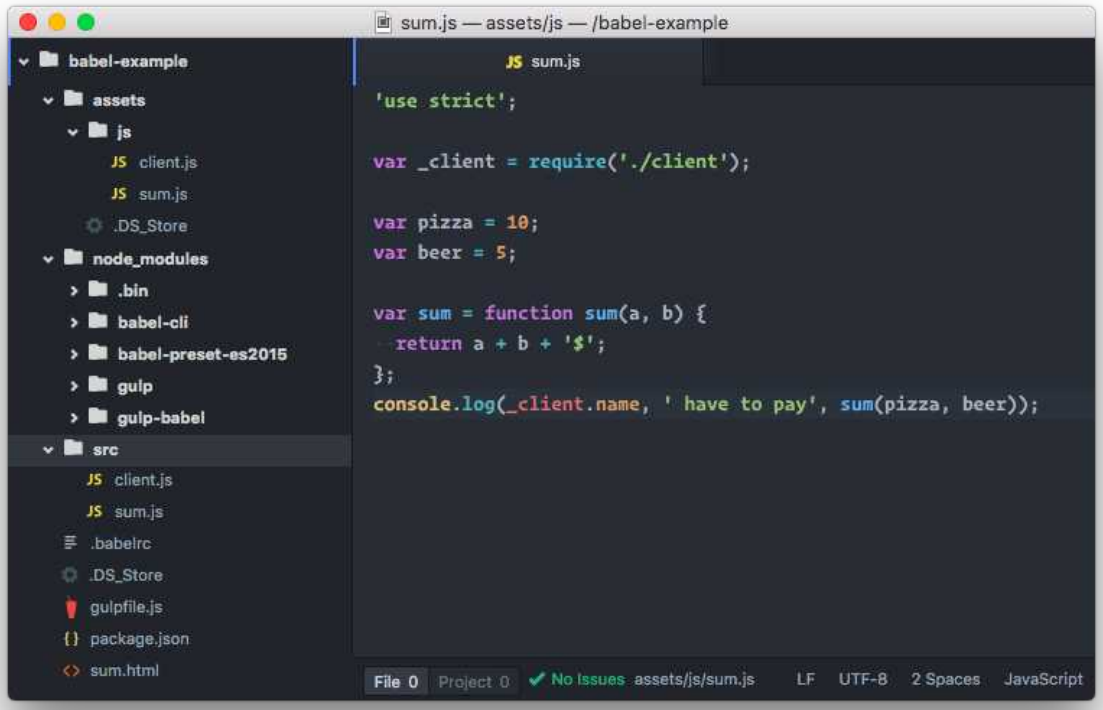
Output of sum.js

You expect here that the same behavior will apply when we open the *html* file in the browser, **but it doesn't**! We get an error instead.



Require is not defined

Check the `node assets/js/sum.js` and notice the `var _client = require('./client');` on top. The reason we get the error in the browser is because **`require()` does not exist in the browser/-client-side JavaScript**. What we have to do is to *bundle* the modules in one file so it can be included within a `<script>` tag.



assets/js/sum.js

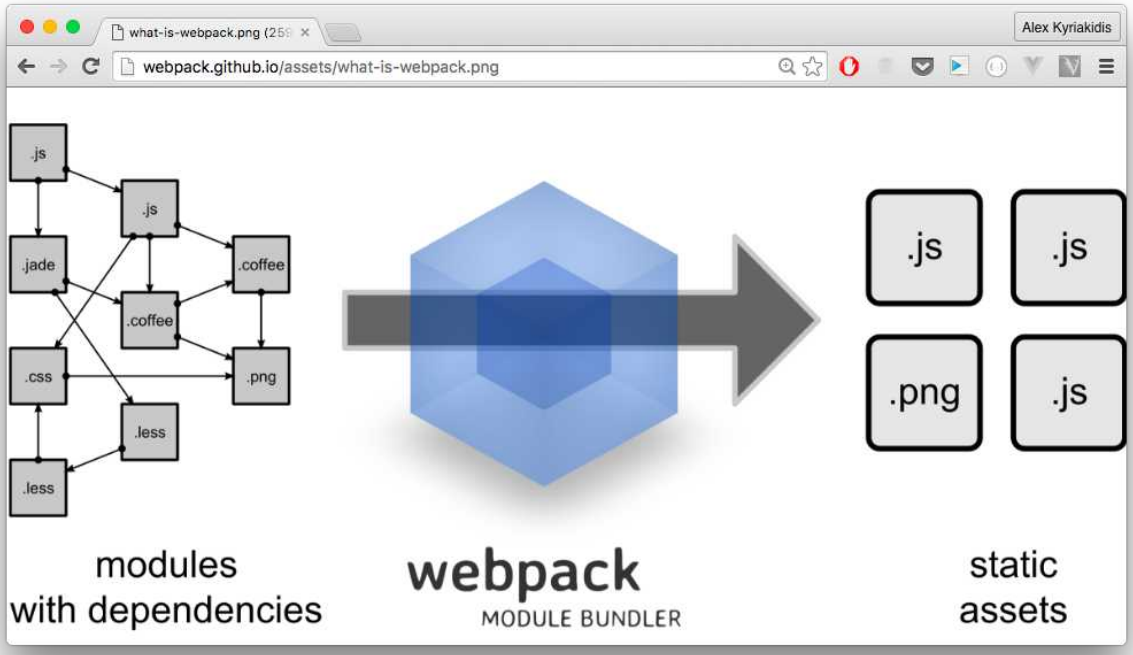
This is where we need **Module Bundlers** like Webpack[10] or Browserify[11].

13.3.2 Webpack

Webpack is a Module Bundler. It takes JavaScript modules, understands their dependencies, and then concatenates them and produces static assets representing those modules.

I'll use Webpack in the next examples. That is because I believe it'll come handy in the future for other uses, since it can do more than bundling modules. With "loaders", we can teach Webpack to transform all type of files in any way we want, before outputting the final bundle.

[10] https://webpack.github.io/
[11] http://browserify.org/



What Webpack does

13.3.3 Installation

I am going to install Webpack globally first, and then add it as dependency in our project.

```
npm install webpack -g

npm install webpack --save-dev
```

Tip

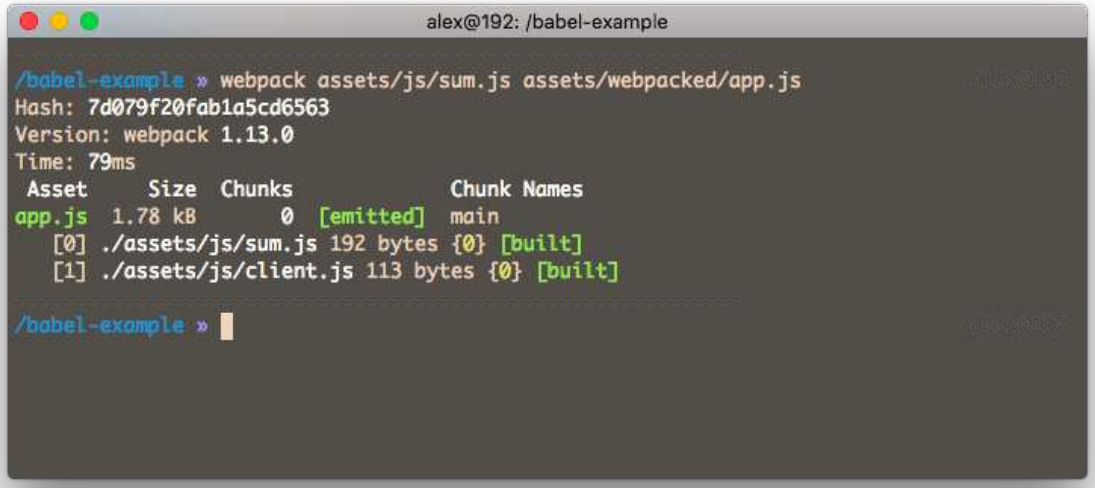
At the time of writing, there is a known bug when you are using Vagrant[12] on Windows, running `npm install` may fail. To solve this issue, exit Vagrant, `cd` your project on your Windows terminal and run `npm install` from there.

13.3.4 Usage

In order to compile our Javascript we have to give Webpack a source point and an output. In our case the source is the babelified 'assets/js/sum.js' and as output I'll set 'assets/webpacked/app.js'.

[12] https://www.vagrantup.com/

```
webpack assets/js/sum.js assets/webpacked/app.js
```



Webpack Output

After the build is completed, we can use the outputed *js, 'assets/webpacked/app.js'*, inside 'sum.html'. We can also run it in the terminal using **`node assets/webpacked/app.js`**.

13.3.5 Automation

If you are lazy, like me, and you don't like having to run `webpack` every time you make a change, you can automate its process. You can configure Webpack to watch your source and rebuild your bundle, when any of your files change. I'll not do it here. Instead, I'm going to integrate it into Gulp, in order to demonstrate how you can combine multiple tools.

Further Learning

If you want to learn more about how Webpack works and how you can configure it, go on and read "Beginner's guide to Webpack"[13] by Nader Dabit[14].

To integrate Webpack into Gulp I'll use a plugin called webpack-stream[15].

[13] https://medium.com/@dabit3/beginner-s-guide-to-webpack-b1f1a3638460
[14] https://twitter.com/dabit3
[15] https://www.npmjs.com/package/webpack-stream

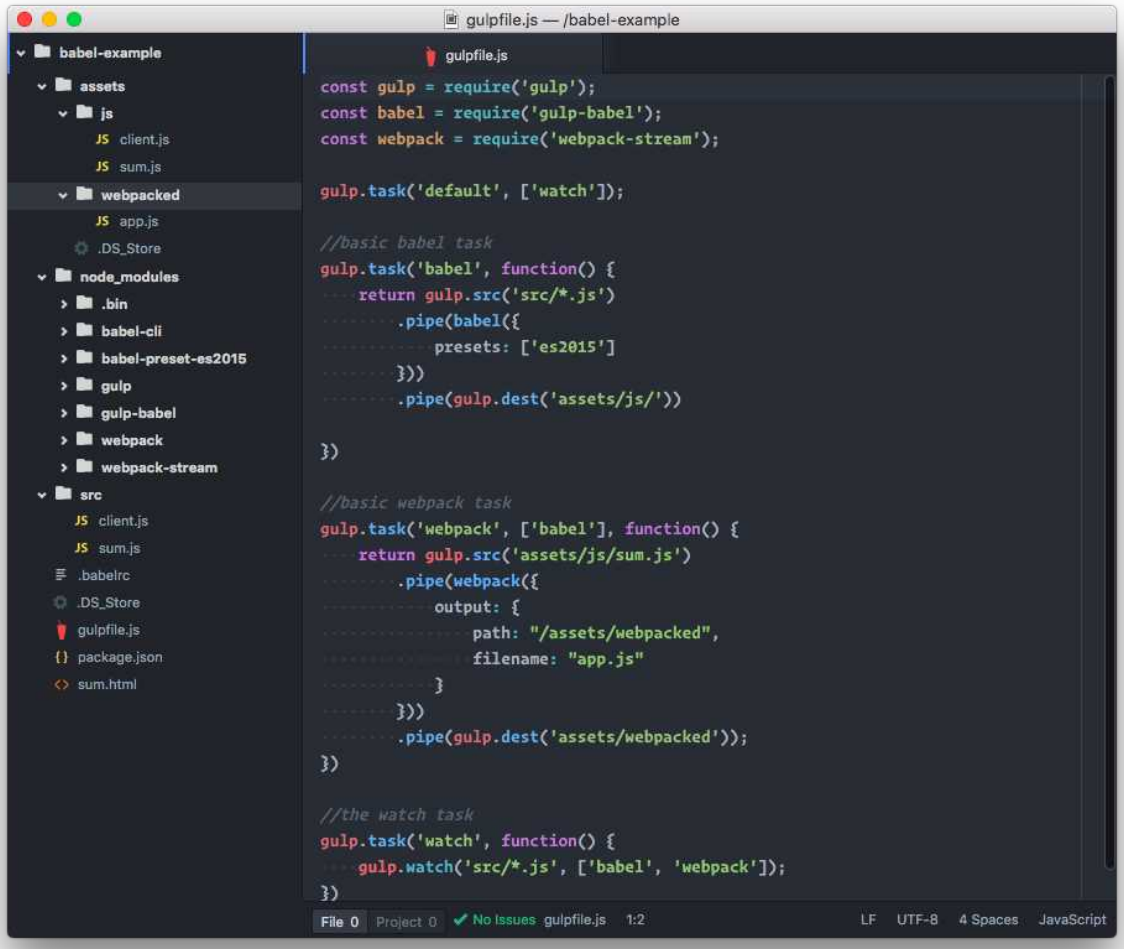
```
>_  npm install webpack-stream
```

After installing, I'll create a new task with the name of 'webpack' and tell Gulp to run it every time it detects a change, immediately after running the 'babel' task.

gulpfile.js

```
const gulp = require('gulp');
const babel = require('gulp-babel');
const webpack = require('webpack-stream');

gulp.task('default', ['watch']);

//basic babel task
gulp.task('babel', function() {
    return gulp.src('src/*.js')
        .pipe(babel({
            presets: ['es2015']
        }))
        .pipe(gulp.dest('assets/js/'))

})

//basic webpack task
gulp.task('webpack', ['babel'], function() {
    return gulp.src('assets/js/sum.js')
        .pipe(webpack({
            output: {
                path: "/assets/webpacked",
                filename: "app.js"
            }
        }))
        .pipe(gulp.dest('assets/webpacked'));
})

//the watch task
gulp.task('watch', function() {
    gulp.watch('src/*.js', ['babel', 'webpack']);
})
```

This solution is not ideal. It is just a demonstration of how you can bind whatever you've learned together. When in production, there are a lot better ways to automate your webpack tasks.



Webpack in Gulp

13.4 Summary

When you want to compile ES6 you can use **Babel**[16].

To automate operations like this and many others (such as minifying, compiling SASS/LESS, etc) you need task runners like **Gulp**[17] or **Grunt**[18].

To bundle things up you can use **Webpack**[19] or **Browserify**[20].

[16] http://babeljs.io/
[17] http://gulpjs.com/
[18] http://gruntjs.com/
[19] https://webpack.github.io/
[20] http://browserify.org/

In the next chapter we will dive into Vue's Single File Components and use several tools that Vue provides you with, along with the tools we have learned.

Note

If you found this chapter hard to understand, don't worry. You don't need to remember all these things. This was just a demonstration in order to give you a better understanding of how things work. From the next chapter we will use "*project-templates*" where things like **Webpack**, **build on change** and much more, are already implemented and we are just taking advantage of it.

14. Mastering Single File Components

As we promised, in this chapter we are going to review Single File Components. To build these single-file Vue components we need tools like **Webpack** with **vue-loader**, or **Browserify** with **vueify**. For our examples, we are going to use Webpack, which we've already seen how it works. If you prefer Browserify or something else, feel free to use it.

Single File Components or Vue Components allow us to specify a template, a script, and style rules, all in one file using `*.vue` extension. So, each component encapsulates its CSS styles, template and JavaScript code, in the same place. And that's were webpack steps in, to bundle this new type of files with the others.

In addition, we need vue-loader[1] to transform Vue components into plain JavaScript modules. *vue-loader* is a loader for Webpack that also provides a very nice set of features, such as ES2015 enabled by default, scoped CSS for each component, and more.

14.1 The `vue-cli`

To avoid configuring Webpack and creating a new workflow from nothing, we will use `vue-cli`.

Info

vue-cli[2] is a simple CLI for scaffolding Vue.js projects. It is a really handy tool, created by **Evan You**.

This awesome tool is the fastest way to get up a pre-configured build. It offers templates with hot-reload, lint-on-save, unit testing, and much more. Currently, it offers scaffold templates for webpack and browserify, but if needed, you can create you own[3].

14.1.1 Vue's Templates

What CLI does, is pulling down templates from Vue.js official templates repository[4] where there are 5 templates available now. I believe this number will grow in the near future. You can check what they include on GitHub.

[1] https://github.com/vuejs/vue-loader
[2] https://github.com/vuejs/vue-cli
[3] https://github.com/vuejs/vue-cli#custom-templates
[4] https://github.com/vuejs-templates

All templates contain a `package.json` file, which handles the project's dependencies and comes with a preset of NPM scripts.

Using Vue's project templates, you get a lot of features together. For example, the "webpack" template's description says that it is *"A full-featured Webpack + vue-loader setup with hot reload, linting, testing & css extraction."*

14.1.2 Installation

We're going to stick with the Webpack setup approach and install **`vue-cli`** using the following command.

```
npm install -g vue-cli
```

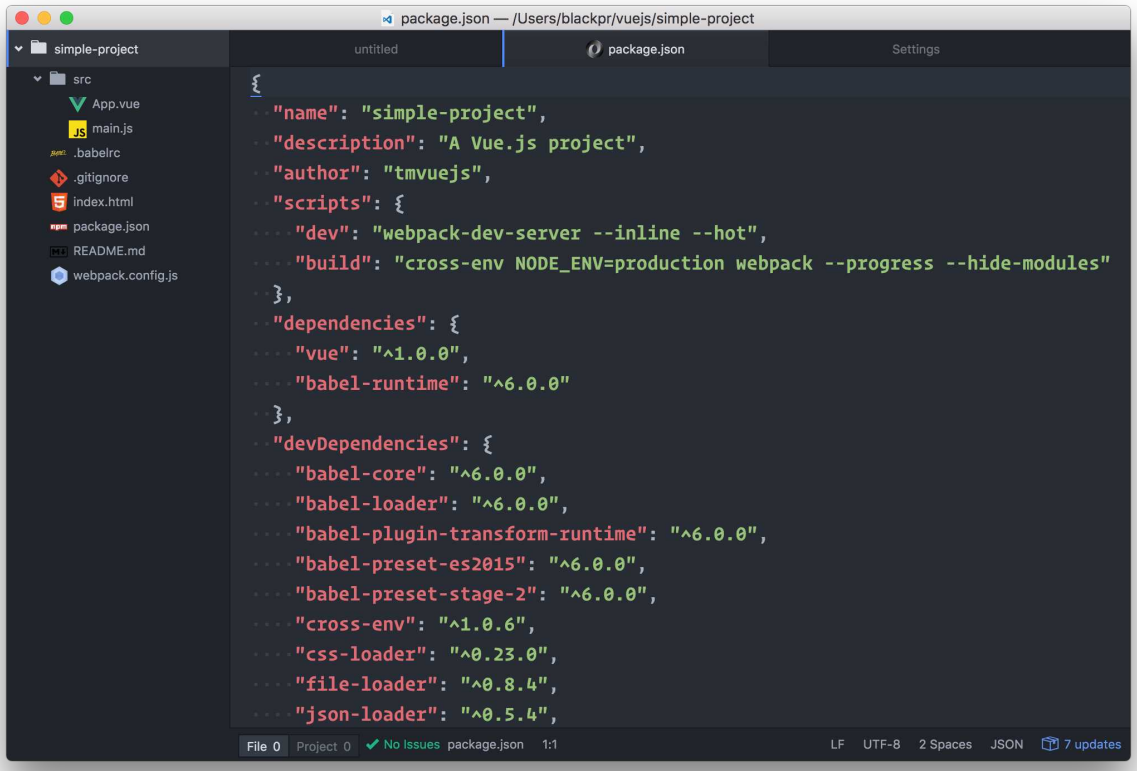
14.1.3 Usage

To use the CLI you have to run the command **`vue init <template-name> <project-name>`** where the **`<template-name>`** is the name of the template (either official or custom) and the **`<project-name>`** is the name of the directory/project you are going to create.

So, if you run

```
vue init webpack-simple simple-project
```

you are going to have a directory named `simple-project` with the following structure:



webpack-simple structure

For our example we will use the full featured webpack template, so our command will be like this:

```
vue init webpack stories-classic-project
```

Info

Use `vue list` to see all available official templates.

When you run this to initialize the chosen template, you will be asked about some details of the project you are building, like the name, the version, etc.

At some point you will be asked to **Pick an ESlint preset**. The available options are feross/standard[5] and airbnb/javascript[6].

[5] https://github.com/feross/standard
[6] https://github.com/airbnb/javascript

I created a table to compare the two styles, in order for you to get a better understanding of what rules each style applies.

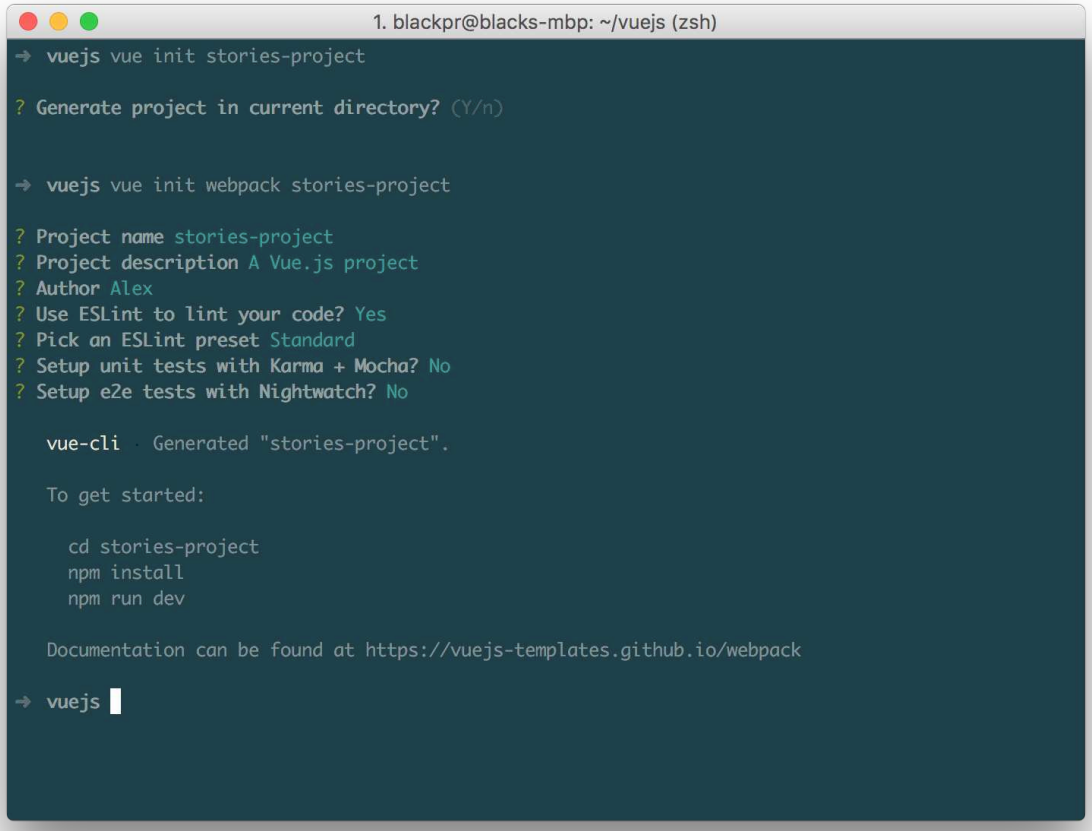
Rules	feross/standard	airbnb/javascript
Indentation	2 spaces	2 spaces
Semicolons	No!	Yes
Unused Variables	Not allowed	Not allowed
String's quotes	Single	Single
Use === instead of ==	Yes	Yes
Number of empty lines allowed	1	2
Space after function name	Yes	No
Start a line with [	No	Yes
End files w/ a newline character	Yes	Yes
Trailing commas allowed	No	No

Standard vs Airbnb

The rules of the table are some of many applied in each style. To consider and decide what fits you the best, check their Github repositories.

After you've selected a style, you'll get some prompts about installing several tools like Karma-Mocha[7] and Nightwatch[8]. We are not going to need these tools right now, so answer the questions negative and continue.

[7] https://github.com/karma-runner/karma-mocha
[8] http://nightwatchjs.org/



Vue's Template Installation

Info

Karma is a plugin, adapter for Mocha[9] testing framework.

Nightwatch enables writing browser automated testing, that run against a Selenium[10] server.

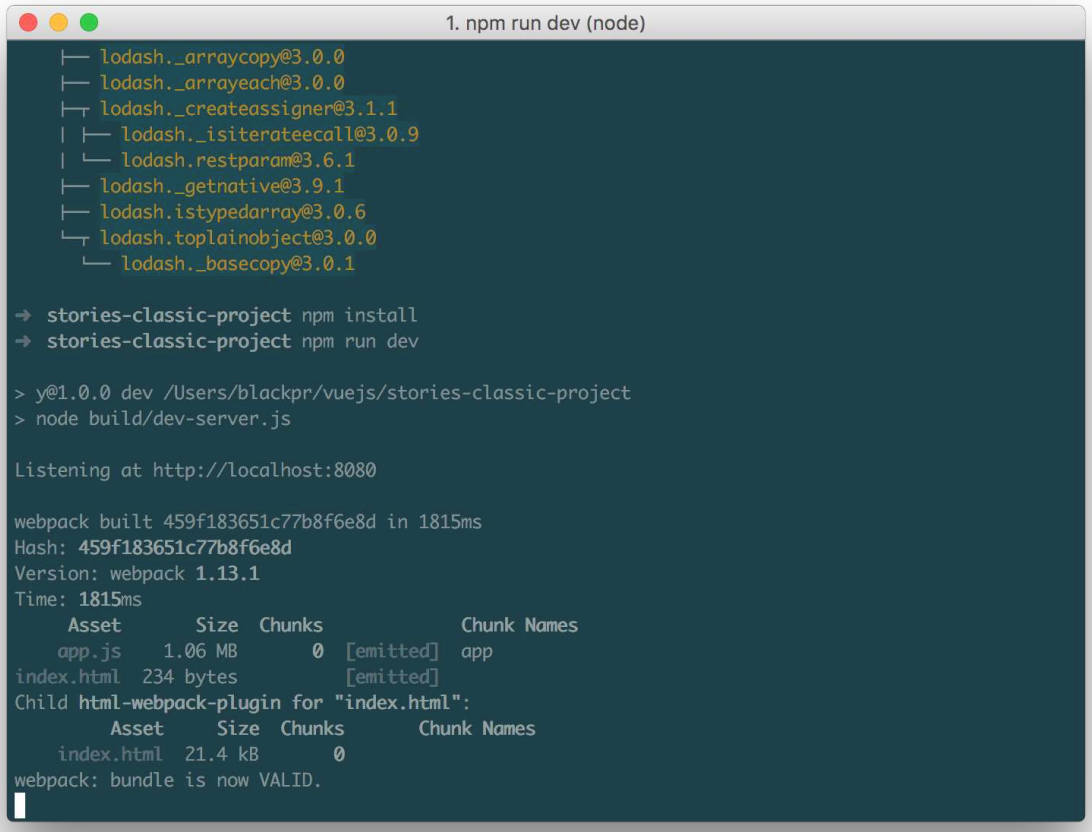
14.2 Webpack Template

To complete the setup of our project we need to install its dependencies. Let's move on and run:

[9] https://mochajs.org/
[10] http://www.seleniumhq.org/

```
cd stories-classic-project

npm install

npm run dev
```

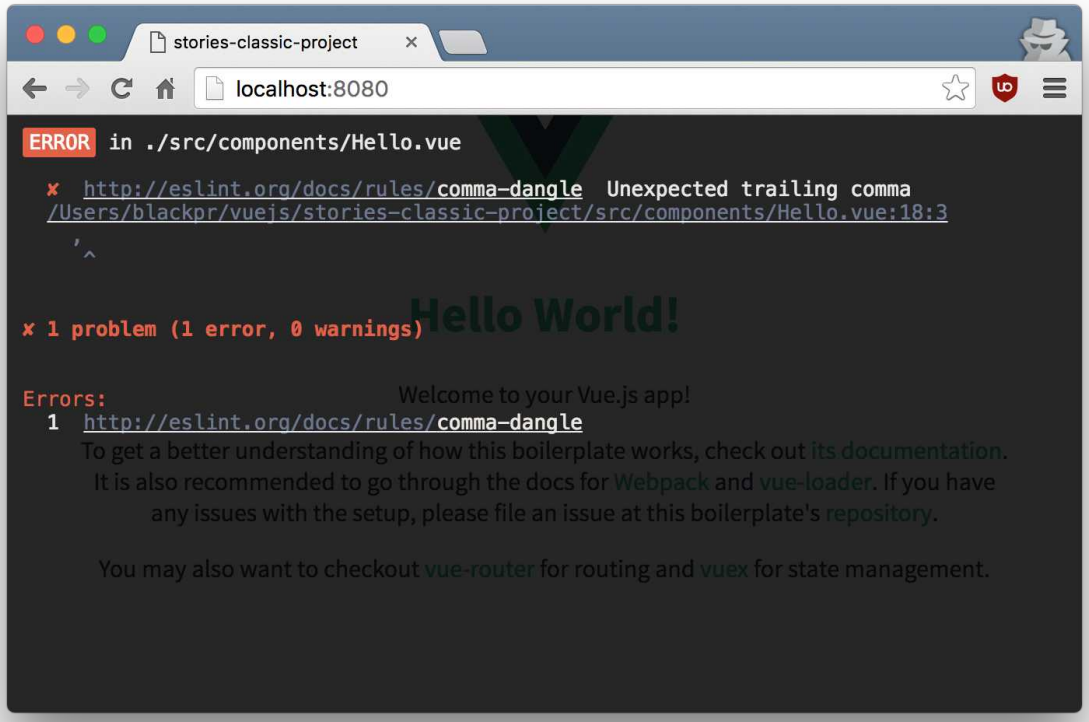
The terminal outputs `Listening at http://localhost:8080`. You should wait until the `webpack: bundle is now VALID` message is posted. Then you are lit!



Server Running...

Warning

When you are using **Webpack Template** you get nice things, such as hot-reload, warnings for errors on-save, etc. But be careful, you have to be explicit in your code. Otherwise you will get errors for extra empty lines between blocks, trailing spaces, indentation other than 2 spaces and other stuff that don't follow the selected style's rules.



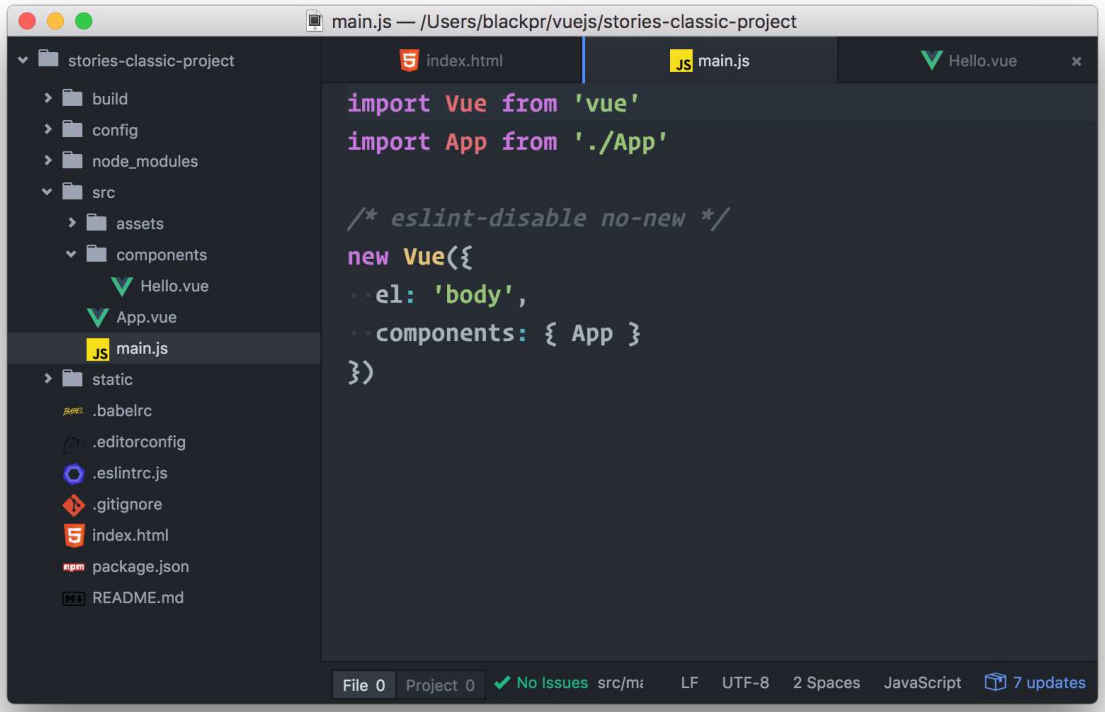
Error overlay

Note

If you use **`webpack-simple`** you will still have the basic features, but the error overlay won't display in the browser, so check the terminal for any errors.

14.2.1 Project Structure

After completing the above steps you should have a project directory filled with all the necessary files.



Webpack structure

The files that you are usually going to play with, are:

1. **`index.html`**
2. **`main.js`**
3. files under **`src`** and **`src/components`** directories

14.2.2 index.html

Lets start with the **`index.html`**. It should look like this

index.html

```
<html>
  <head>
    <meta charset="utf-8">
    <title>stories-classic-project</title>
  </head>
  <body>
    <app></app>
    <!-- built files will be auto injected -->
  </body>
</html>
```

As you can see, it is a pretty basic setup with a single component already included. The comment refers to the script, **`app.js`**, which is the output of Webpack. It basically means that after Webpack has bundled the scripts, it will automatically inject the outputted `script` here, so that you don't have to include it manually.

14.2.3 Hello.vue

If you are following along, navigate to **`src/components`**, open **`Hello.vue`** file and take a look how a **`*.vue`** file is.

src/components/Hello.vue

```
<template>
  <div class="hello">
    <h1>{{ msg }}</h1>
  </div>
</template>

<script>
export default {
  data () {
    return {
      // note: changing this line won't causes changes
      // with hot-reload because the reloaded component
      // preserves its current state and we are modifying
      // its initial state.
      msg: 'Hello World!'
    }
  }
}
```

```
</script>

<!-- Add "scoped" attribute to limit CSS to this component only -->
<style scoped>
h1 {
  color: #42b983;
}
</style>
```

Notice here the component's template, script, and style rules. Inside the **<script>** tag, the components contains only its **data**. There is no need to define a template. It will be binded automatically **if** <template> block exists. The **<template>** block defines the template of the component, of course. Think of **<template>** like it is a **<template-hello>** tag in your HTML. We could have only the **<script>** block, but this way, we wouldn't take advantage of **.vue** file (Single File Component) benefits.

Remember that **export**, ES6 feature, is handled by those nice tools (transpilers + module bundlers) we have installed.

Info

ES6 allows us to have any numbers of exports, but in single file components that's not the case. If you try to do additional exports you will get a warning similar to this: **[vue-loader] src/components/Hello.vue: named exports in *.vue files are ignored.**

The **<style>** block, defines the CSS styles as expected. Read the provided comments carefully, especially the first one, as I myself have regretfully neglected it!

Note

Although we have hot-reload enabled, as the comment states, changing the lines within our returned data won't update because it preserves its current state and we are modifying its initial state. So a page refresh is required to see the changes we have made.

14.2.4 App.vue

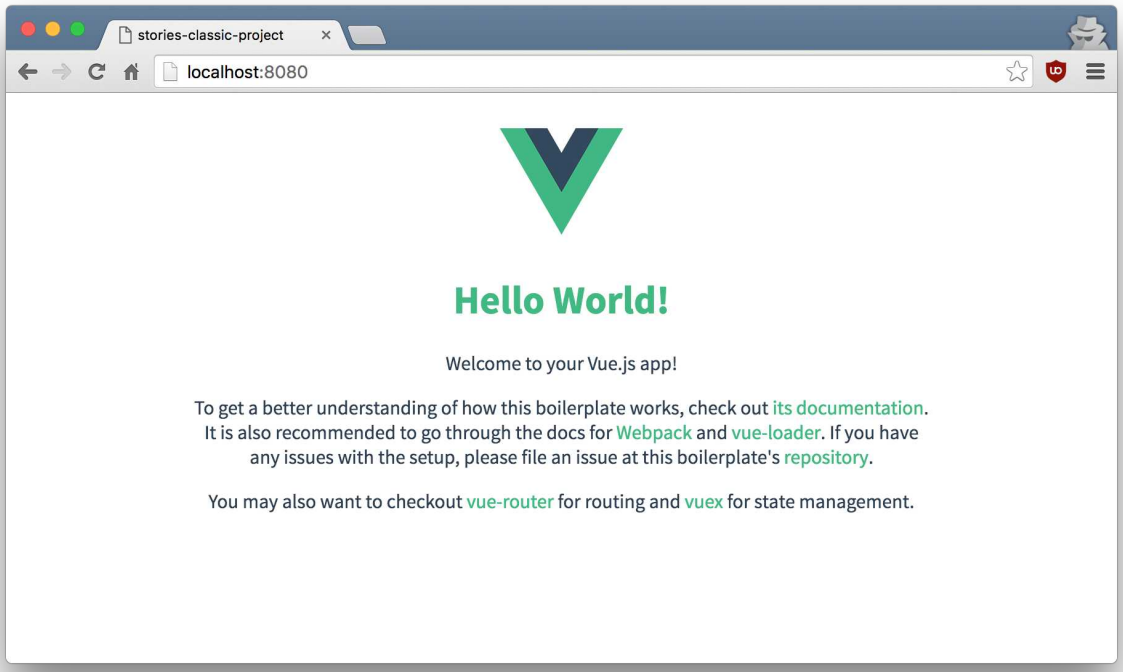
The **App.vue** file is located in the **src** directory and is the one which contains the main template of the application. This component is usually responsible to include the other components.

App.vue *has a few more lines with texts and styles, but since we are focusing on the structure, we have shortened it a little.*

src/App.vue

```
<template>
  <div id="app">
    <img class="logo" src="./assets/logo.png">
    <hello></hello>
    <p>
      Welcome to your Vue.js app!
    </p>

    ...

  </div>
</template>

<script>
import Hello from './components/Hello'

export default {
  components: {
    Hello
  }
}
</script>

<style>
...
</style>
```

It has the same structure as the **`Hello.vue`** file we saw above. By default, there is a **`components`** object which contains the **`Hello`** component. Inside this object, we will import any new components. In the template, there is the **`<hello></hello>`** tag, and therefore the template of the `Hello` component will be displayed.



Project's Homepage

14.2.5 main.js

The **`main.js`** file within **`src`**, as you imagine, is our main script.

src/main.js

```
1  import Vue from 'vue'
2  import App from './App'
3
4  /* eslint-disable no-new */
5  new Vue({
6    el: 'body',
7    components: { App }
8  })
```

It imports **`Vue`** as a module from **`node_modules`** and **`App`** component from the **`src`** directory as well. Below is our **`Vue`** instance and in the components object, there is the **`App`** one.

Whenever you need to import a script or a component globally, you can put it within **`main.js`**.

Info

You can find more information about the Project Structure of Webpack template on its documentation[11]

14.3 Forming `.vue` Files

We have seen how a single file component looks like and how it is used in a project. It is time to create a few, in a real-life scenario. Assume we want to create some kind of social network or forum, where users post their stories and experiences. To create our startup, we are going to need 2 forms, one for registration and login, and one page to display users' stories.

Before we begin, we'll include Bootstrap globally in order to be able to use its styles within all our components. To do so, we have to update our **`index.html`**.

index.html

```
<html>
  <head>
    <meta charset="utf-8">
    <title>stories-classic-project</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6\
/css/bootstrap.min.css">
  </head>
  <body>
    <app></app>
    <!-- built files will be auto injected -->
  </body>
</html>
```

Now that we have Bootstap installed globally, let's create the login form, in a new **`Login.vue`** file.

[11] http://vuejs-templates.github.io/webpack/structure.html

src/components/Login.vue

```
<template>
      <h2>Sign in</h2>
      <input type="email" placeholder="Email address">
      <input type="password" placeholder="Password">
      <button class="btn">Sign in</button>
</template>

<script>
export default {
  created () {
    console.log('login')
  }
}
</script>
```

And there it is. In order to view the file in the browser we have to include our `Login` component somewhere. So, we'll import it into the main 'App.vue' file and append it to its components object.

src/App.vue

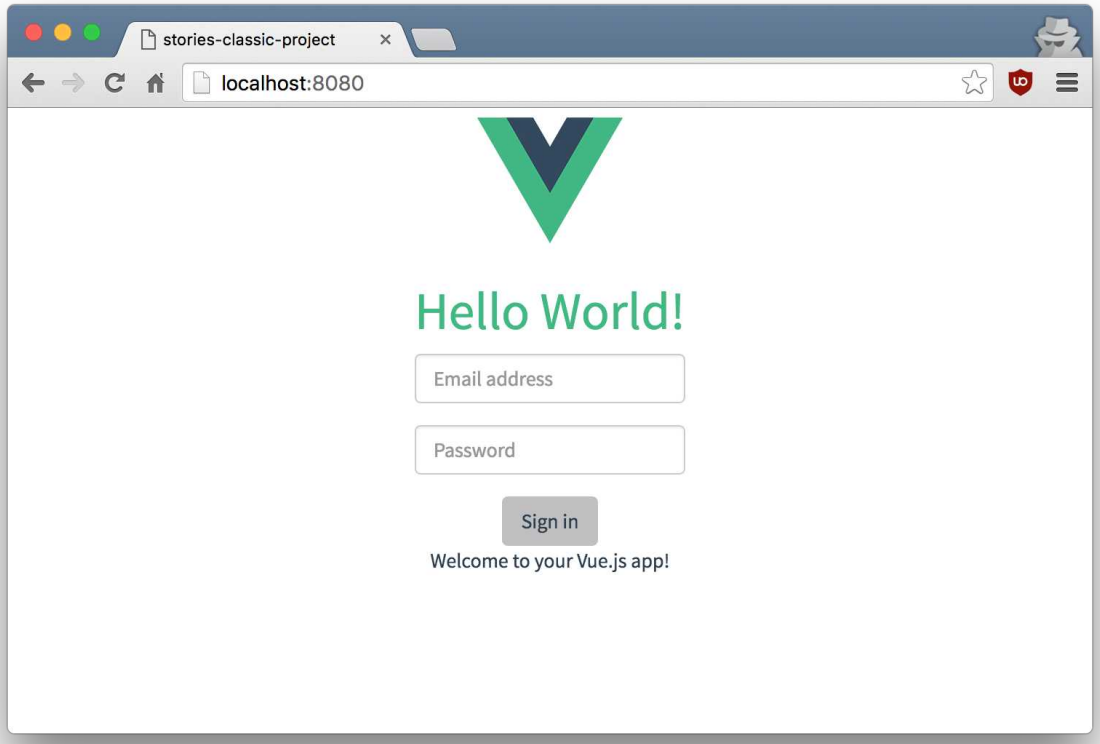
```
<template>
  <div id="app">
    <img class="logo" src="./assets/logo.png">
    <hello></hello>
    <p>
      Welcome to your Vue.js app!
    </p>
    ...
  </div>
</template>

<script>
import Hello from './components/Hello'
import Login from './components/Login'

export default {
  components: {
    Hello,
    Login
  }
}
```

```
</script>

<style>
...
</style>
```

If you hit reload in your browser you won't see the "Login" component under the pre-existing component **`Hello`** yet, because we also need to reference it. Place it under the '<hello></hello>' and you will have a nice login form!

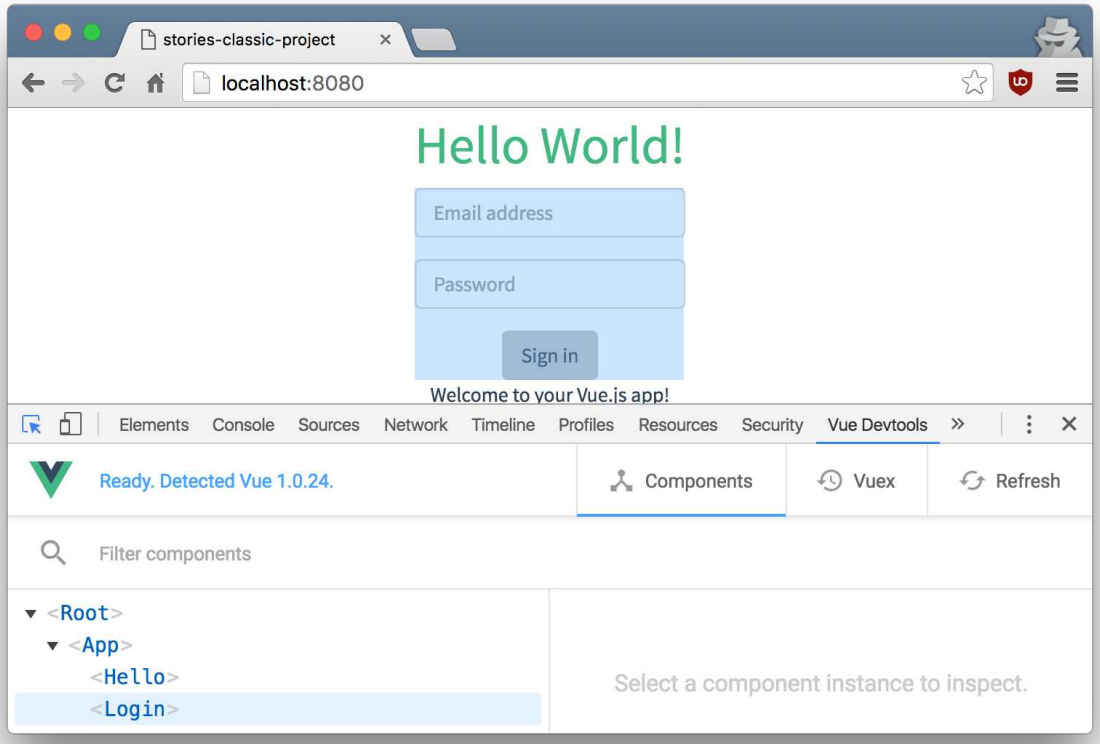
src/App.vue

```
<template>
  <div id="app">
    ...
    <hello></hello>
    <login></login>
    ...
</template>
...
...
```



Login Component

If you open the browser console, you should see the `login` message we are logging when the Component is created. If you are using Vue-devtools, which is highly recommended, you should also see it in the components tree view.



Tree View

Let's create another Component, this time for registration.

src/components/Register.vue

```
<template>
  <h2>Register Form</h2>
  <input placeholder="First Name" class="form-control">
  <input placeholder="Last Name" class="form-control">
  <input placeholder="Email address" class="form-control">
  <input placeholder="Pick a password" class="form-control">
  <input placeholder="Confirm password" class="form-control">
  <button class="btn">Sign up</button>
</template>

<script>
export default {
  created () {
    console.log('register')
```

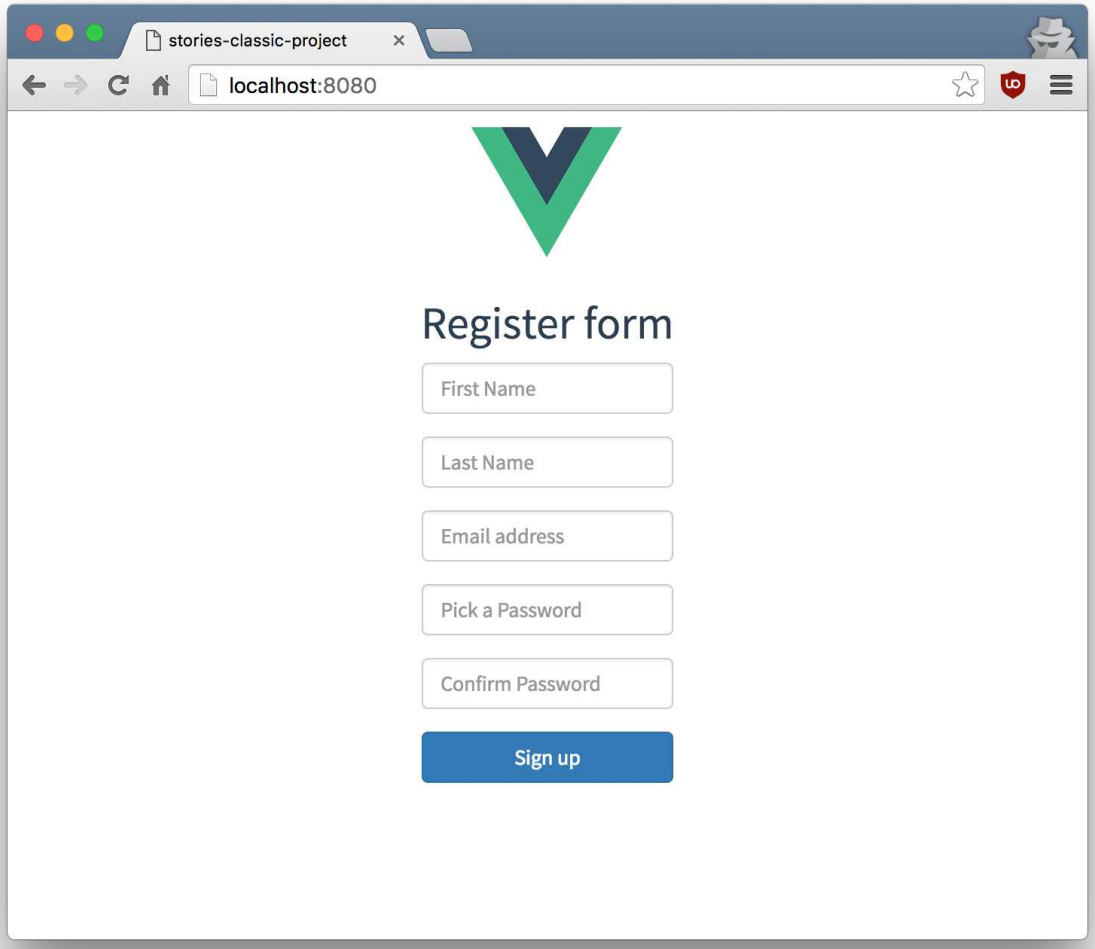
```
  }
}
</script>
```

Then we can import it in the **App.vue** file.

src/App.vue

```
<template>
  <div id="app">
    ...
    <!-- <hello></hello> -->
    <!-- <login></login> -->
    <register></register>
    ...
  </div>
</template>

<script>
// import Hello from './components/Hello'
// import Login from './components/Login'
import Register from './components/Register'

export default {
  components: {
    // Hello,
    // Login,
    Register
  }
}
</script>
...
...
```

The **Register** component's template appears, when we check the browser.



Register Component

Note

The other components are commented out because we don't want to display them one under the other. The `Hello` component is there by default, but we are not going to use it in any further examples, so we will remove it.

We said that we are working on a social network (or something relevant), so we want a place to display the stories. Thus, we are going to create a **`Stories`** component which when imported, brings up all the stories told by the users.

src/components/Stories.vue

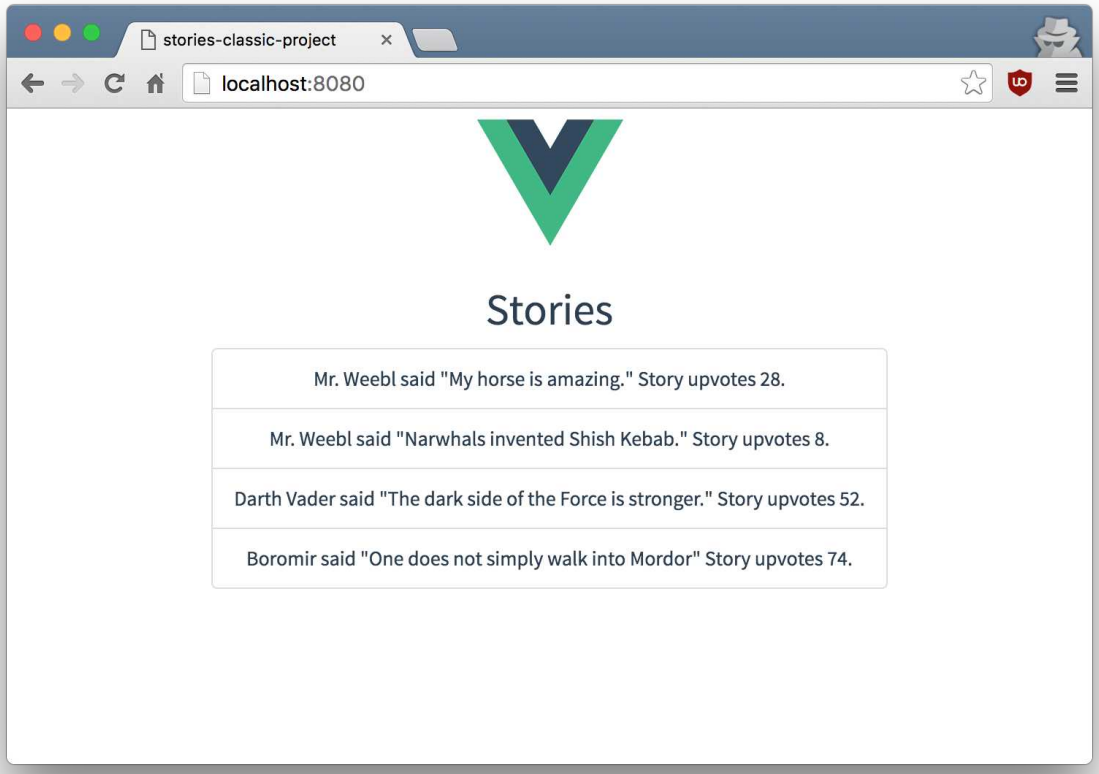
```
<template>
  <ul class="list-group">
    <li v-for="story in stories"  class="list-group-item">
      {{ story.writer }} said "{{ story.plot }}"
      Story upvotes {{ story.upvotes }}.
    </li>
  </ul>
</template>

<script>
  export default {
    data () {
      return {
        stories: [
          {
            plot: 'My horse is amazing.',
            writer: 'Mr. Weebl',
            upvotes: 28,
            voted: false
          },
          {
            plot: 'Narwhals invented Shish Kebab.',
            writer: 'Mr. Weebl',
            upvotes: 8,
            voted: false
          },
          {
            plot: 'The dark side of the Force is stronger.',
            writer: 'Darth Vader',
            upvotes: 52,
            voted: false
          },
          {
            plot: 'One does not simply walk into Mordor',
            writer: 'Boromir',
            upvotes: 74,
            voted: false
          }
        ]
      }
    }
```

```
  }
</script>
```

This is the **Stories.vue** file. We can use it in our main **App.vue** file. At this point the stories are hard coded for simplicity. Time to import it just like the other components.

src/App.vue

```
<template>
  <div id="app">
    ...
    <!-- <login></login> -->
    <!-- <register></register> -->
    <stories></stories>
    ...
  </div>
</template>

<script>
// import Login from './components/Login'
// import Register from './components/Register'
import Stories from './components/Stories'

export default {
  components: {
    // Login,
    // Register,
    Stories
  }
}
</script>
```



Stories Component

Great! Now we have a page to display all the listings.

14.3.1 Nested Components

We would like to be able to display the most "famous" stories, at any place we want to. So after the creation of a *'Famous'* component, we should be able to use it anywhere.

Let's create the component.

src/components/Famous.vue

```
<template>
<h2>Trending stories<strong>({{famous.length}})</strong></h2>
  <ul class="list-group">
    <li v-for="story in famous"  class="list-group-item">
      {{ story.writer }} said "{{ story.plot }}".
      Story upvotes {{ story.upvotes }}.
    </li>
  </ul>
</template>

<script>
export default {
  computed: {
    famous () {
      return this.stories.filter(function (item) {
        return item.upvotes > 50
      })
    }
  },
  data () {
    return {
      stories: [
        {
          plot: 'My horse is amazing.',
          writer: 'Mr. Weebl',
          upvotes: 28,
          voted: false
        },
        {
          plot: 'Narwhals invented Shish Kebab.',
          writer: 'Mr. Weebl',
          upvotes: 8,
          voted: false
        },
        {
          plot: 'The dark side of the Force is stronger.',
          writer: 'Darth Vader',
          upvotes: 52,
          voted: false
        },
        {
```

```
          plot: 'One does not simply walk into Mordor',
          writer: 'Boromir',
          upvotes: 74,
          voted: false
        }
      ]
    }
  }
}
</script>
```

This is the whole **Famous.vue** file. We have **filtered** the `stories` array using computed properties, as we saw in previous chapters, and created a **template** to display them.

Note

Stories array is hard coded again here and data are the same as before. This is a bad practice, we will find a way later to define `stories` array once and use it within all components.

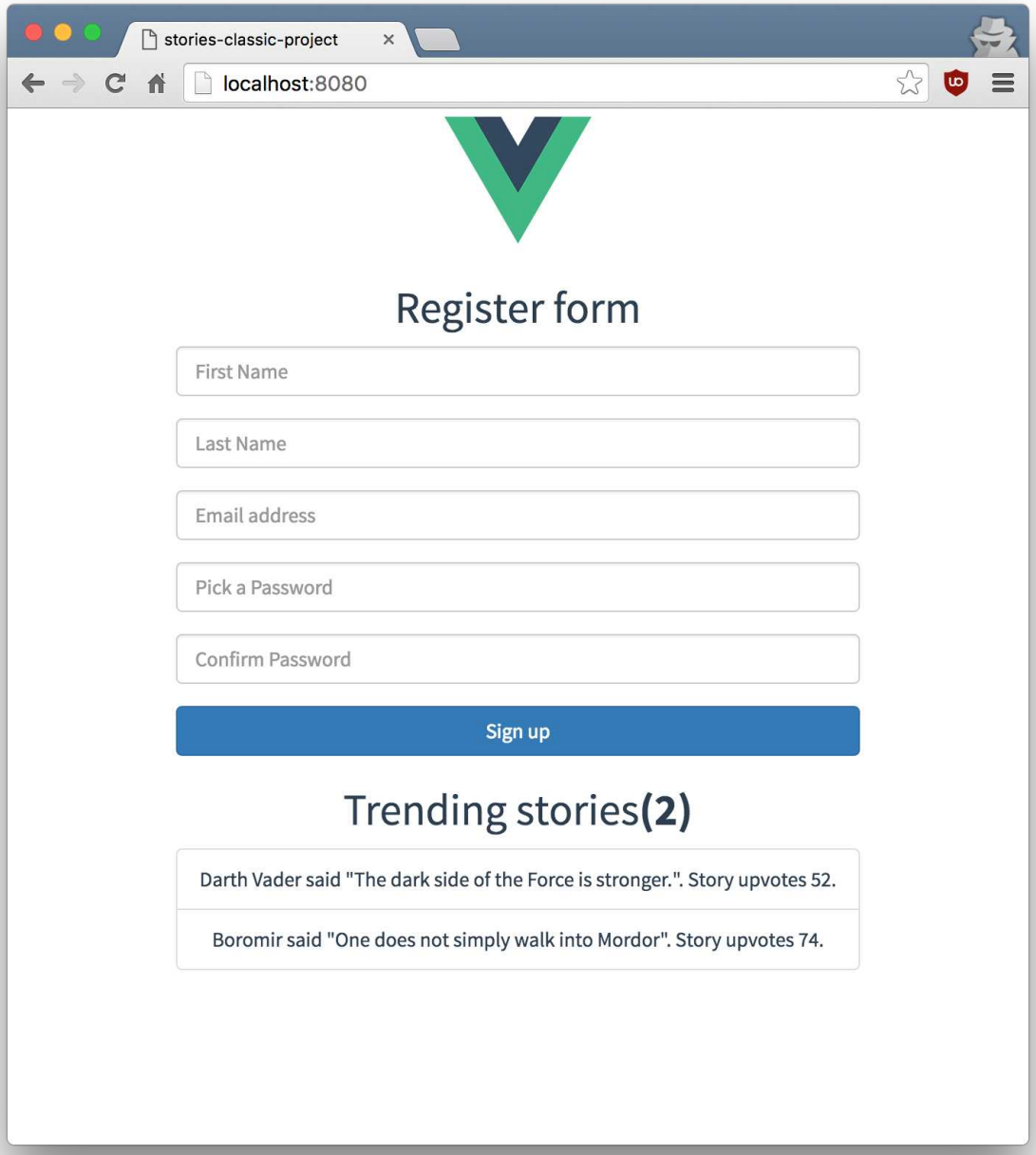
But where could we use this component? An idea is to have it within the registration page, so the user could read the most trending stories and become intrigued. This means - in the current project - that we need to have the **Famous** component within the **Register** one. Well, this can be done the same way we did it inside **App.vue**.

So, open **Register.vue**, import it there, and reference it within the template.

src/components/Register.vue

```
<template>
  <h2>Register Form</h2>
    ...
    ...
  <famous></famous>
</template>
<script>
import Famous from './Famous'

export default {
  components: {
    Famous
  },
  created () {
```

```
15      console.log('register')
16    }
17  }
18  </script>
```



Registration page with top stories

Pay attention to the import file path. Now that the two files are in the same directory, you have to use ‘./Famous’ instead of the full path. This is an easy mistake to make, especially if you’re not familiar with it!

14.4 Eliminating Duplicate State

In the previous examples we have hardcoded the data (an array of stories) within each Component (`*.vue` file). **This is not a proper way to work with data.**

When more than one Component uses the same data, it's a good practice to create/fetch the array once, and then find a way to share it between application's components.

`Stories.vue` and **`Famous.vue`** are using the same `stories` array. We will review two ways of creating the array once, and then sharing it between the components.

1. Using component properties.
2. Using a global store.

14.4.1 Sharing with Properties

The first thing we are going to do is to move the **`stories`** array to **`App`** component.

src/App.vue

```
<script>
...

export default {
  components: {
  ...
  },
  data () {
    // here we place the stories array
    return {
      stories: [
        {
          plot: 'My horse is amazing.',
          writer: 'Mr. Weebl',
          upvotes: 28,
          voted: false
        },
        {
          plot: 'Narwhals invented Shish Kebab.',
          writer: 'Mr. Weebl',
          upvotes: 8,
          voted: false
        },
```

```
24          {
25            plot: 'The dark side of the Force is stronger.',
26            writer: 'Darth Vader',
27            upvotes: 52,
28            voted: false
29          },
30          {
31            plot: 'One does not simply walk into Mordor',
32            writer: 'Boromir',
33            upvotes: 74,
34            voted: false
35          }
36        ]
37      }
38    }
39  }
40  </script>
```

The next step is to remove the **`data()`** from **`Stories`** and **`Famous`** components, and declare `stories` property.

Let's do it for the first component.

src/components/Stories.vue

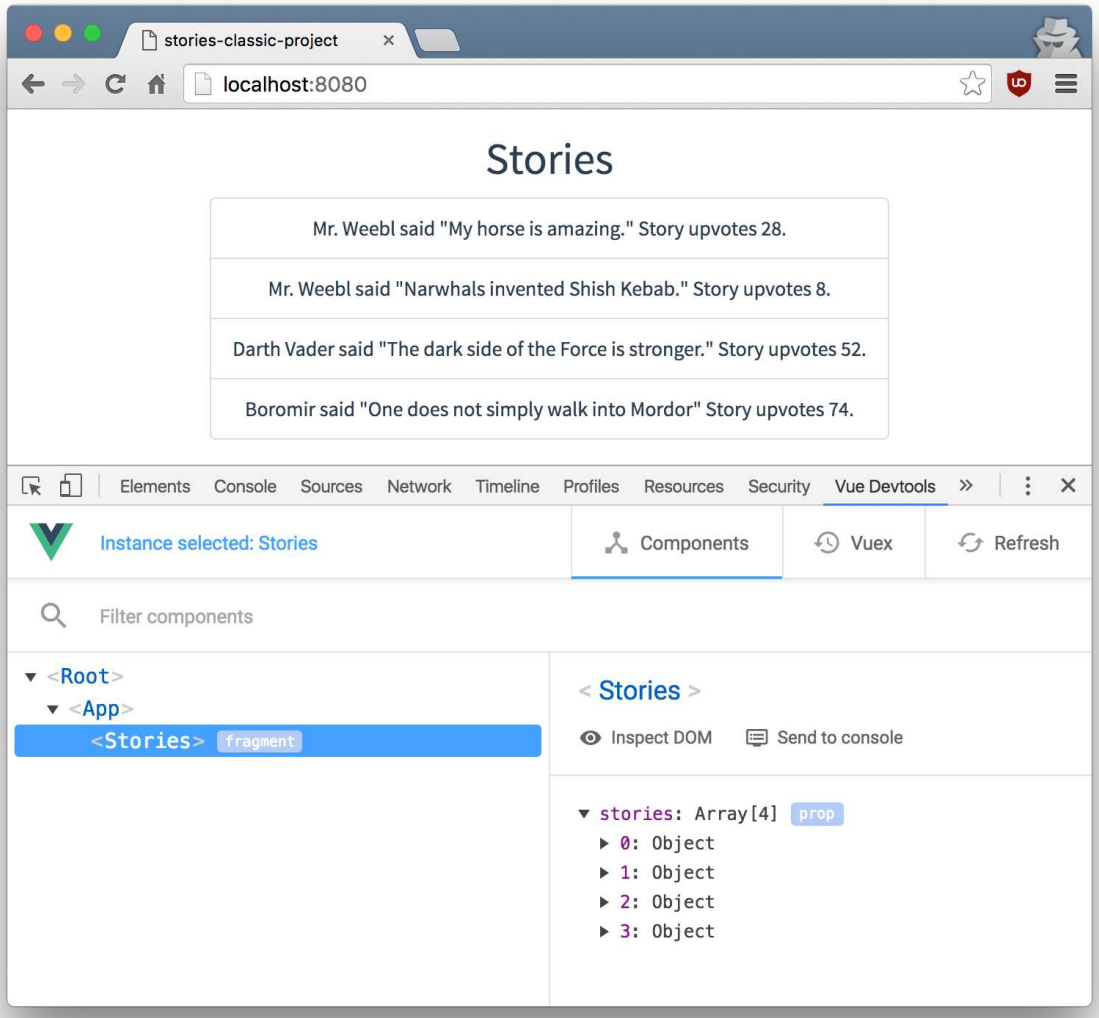
```
1  <script>
2    export default {
3      props: ['stories']
4    }
5  </script>
```

We have to update the way we reference our component within **`App.vue`**.

src/App.vue

```
1  <template>
2    <div id="app">
3      ...
4      <stories :stories="stories"></stories>
5      ...
6      <p>
7        Welcome to your Vue.js app!
8      </p>
9    </div>
10 </template>
```

Here we bind **`stories`** prop to **`stories`** array.



Same output, using props

Success, we got our stories again, fetched from the parent component!

We can't do the same for "famous" component because it is not referenced inside **`App.vue`**. We will have to pass our array to **`Register`** component in order to pass it to **`Famous`**.

src/App.vue

```
<template>
  <div id="app">
    ...
    <register :stories="stories"></register>
    ...
  </div>
</template>
```

src/components/Register.vue

```
<template>
  <h2>Register Form</h2>
  ...
  <famous :stories="stories"></famous>
</template>

<script>
import Famous from './Famous'

export default {
  components: {
    Famous
  },
  props: ['stories']
}
</script>>
```

src/components/Famous.vue

```
<script>
  export default {
    props: ['stories'],

    computed: {
      famous () {
        return this.stories.filter(function (item) {
          return item.upvotes > 50
        })
      }
    }
```

```
  }
</script>
```

This implementation works, but is not efficient, because `Famous` component is **not independent**. This means that we cannot use it wherever we want, unless we pass down the data from root component (`App.vue`).

In a scenario where a not independent component is deeply nested, you will have to pass a useless property, from component to component, just to be able to use it. In our case, if we wanted to use `Famous` inside `Register`'s sidebar's widget, we would have to carry the stories array all the way long.

App -> Register -> Sidebar -> WidgetX -> Famous

14.4.2 Global Store

The "props" way seemed nice at first, but as seen in the **`famous`** component, as a project gets bigger and components get nested into others, data management and sharing between them gets really hard to track.

So let's make data of our examples a bit easier to handle.

We can extract the `stories` data to a **`.js`** file, store them to a **`constant`** and later import them at the desirable locations.

I'll name our **`js`** file `store.js` and put it inside **`/src`** directory.

src/store.js

```
export const store = {
  stories: [
    {
      plot: 'My horse is amazing.',
      writer: 'Mr. Weebl',
      upvotes: 28,
      voted: false
    },
    {
      plot: 'Narwhals invented Shish Kebab.',
      writer: 'Mr. Weebl',
      upvotes: 8,
      voted: false
    },
    {
      plot: 'The dark side of the Force is stronger.',
```

```
      writer: 'Darth Vader',
      upvotes: 52,
      voted: false
    },
    {
      plot: 'One does not simply walk into Mordor',
      writer: 'Boromir',
      upvotes: 74,
      voted: false
    }
  ]
}
```

Warning

The `stories` **`prop`** must be removed from all files, because we have changed the way of data storage and there can be conflicts, which can break our build.

After we have stored all data in **`store.js`** we can import it within **`Stories.vue`** using ES6 modules syntax.

src/components/Stories.vue

```
<script>
    import {store} from '../store.js'

    export default {
      data () {
        return {
          //will give us access to store.stories
          store
        }
      },
      created () {
        console.log('stories')
      }
    }
  </script>
```

Because we are importing the **`store`** object we have to change the component's template as well.

src/components/Stories.vue

```
<template>
  <ul class="list-group">
    <li v-for="story in store.stories"  class="list-group-item">
      {{ story.writer }} said "{{ story.plot }}"
      Story upvotes {{ story.upvotes }}.
    </li>
  </ul>
</template>
```

We are using **`v-for`** to render the items of the array (**`store.stories`**). Our list of stories is displaying as before.

We could do the same thing without having to change the template, by binding component's `stories` attribute to `store.stories` directly.

src/components/Stories.vue

```
<script>
  data () {
    return {
      // Bind directly to stories
      stories: store.stories,
    }
  }
</script>
```

The same thing applies for **`Famous.vue`**.

src/components/Famous.vue

```
<script>
  import {store} from '../store.js'

  export default {
    data () {
      return {
        stories: store.stories
      }
    },
    computed: {
      famous () {
```

```
        return this.stories.filter(function (item) {
          return item.upvotes > 50
        })
      }
    }
  }
</script>
```

If we didn't bind to stories, **`famous()`** computed property would have to be updated to filter **`this.store.stories`**.

Once you get used to work with global objects I believe you are going to **love it!** :)

Code Examples

You can find the code examples of this chapter on GitHub[12].

[12] https://github.com/hootlex/the-majesty-of-vuejs/tree/master/examples/14.%20Mastering%20Single%20File%20Components

15. Swapping Components

Using Singe File Components is the simplest way to build a Single Page Application with Vue. Using them requires to get familiar with many -if not all- features provided. We saw in this chapter how to setup a fresh project, create `*.vue` files and manage duplicate state so far. Now it's time to review a way to swap view-specific components.

For reference in the previous examples, we had 3 components inside `App.vue` and some others nested within. Not very manageable or expandable. Consequently, we need to practice ways on how to swap components dynamically, so we won't need to fill a page with them. Lets see some examples.

15.1 Dynamic Components

15.1.1 component is

One way to surpass this, is to use the reserved `<component>` element and use the same mount point to dynamically switch between multiple components, by using the `is` attribute.

src/App.vue

```
<template>
  <div id="app">
    <component is="hello"></component>
    <p>
      This is very useful...
    </p>
  </div>
</template>

<script>
import Hello from './components/Hello'
// Component Hello returns a template containing a "msg" property of data
export default {
  components: {
    Hello
  }
}
</script>
```

The **Hello.vue** file comes with the default files and folders of the project. Now we have exact the same results we had before our changes, but here we are using the **<component>** element. To see how this would work dynamically, check the next example where we are changing between 2 different single components by clicking on their links.

First create another component, like **Hello.vue** in structure, but with a different message, named **Greet.vue**.

src/Greet.vue

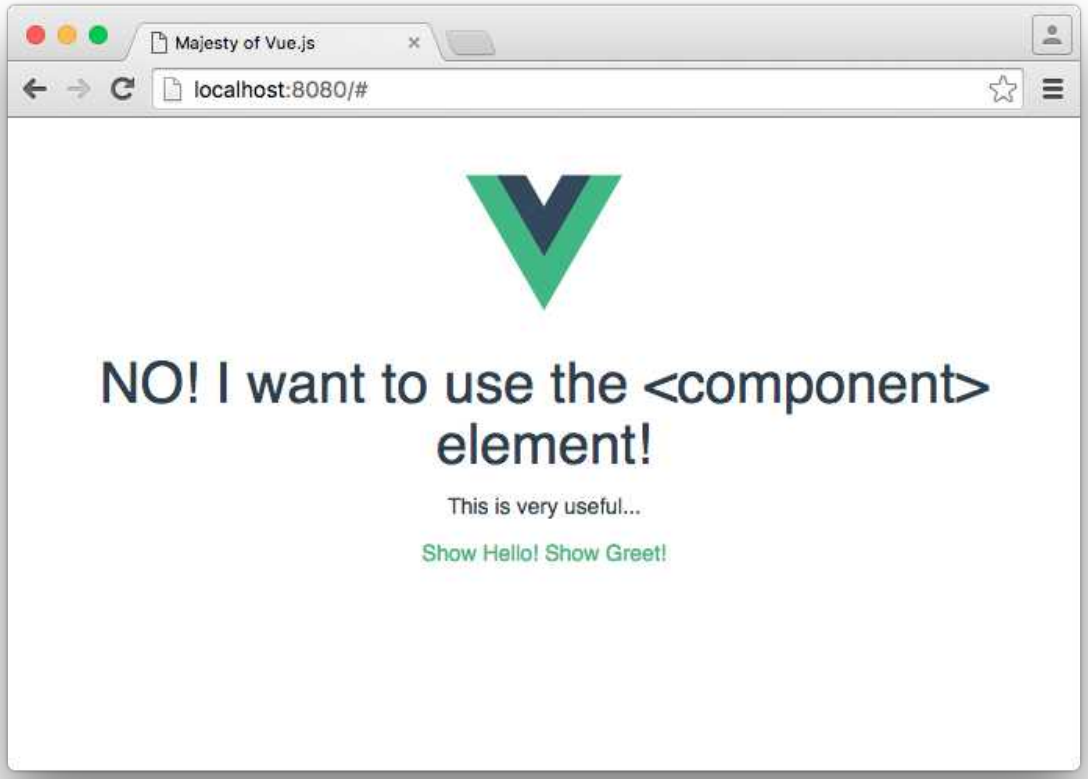
```
<template>
  <div class="greet">
    <h1>{{ msg }}</h1>
  </div>
</template>

<script>
export default {
  data () {
    return {
      msg: 'No! I want to use the <component> element!'
    }
  }
}
</script>
```

We made **Greet** display a message to manifest its presence! Let's import it in **App.vue** and give the user the ability to swap between the 2 components.

src/App.vue

```
<template>
  <div id="app">
    <img class="logo" src="./assets/logo.png">
    <component :is="currentComponent">
            <!-- component changes when this.currentComponent changes! -->
    </component>
    <p>
      This is very useful...
    </p>
    <a href="#" @click="currentComponent = 'hello'">Show Hello!</a>
    <a href="#" @click="currentComponent = 'greet'">Show Greet!</a>
  </div>
</template>
```

```
14
15  <script>
16  import Hello from './components/Hello'
17  import Greet from './components/Greet'
18
19  export default {
20    components: {
21      Hello,
22      Greet
23    },
24    data () {
25      return {
26        currentComponent: 'greet'
27      }
28    }
29  }
30  </script>
```



Greet.vue

Well, as you can see, we are binding the special attribute `is` to `currentComponent`, so when its value changes, the displaying component will also change. To swap the view, the user just have to click on either link to change the value of our data property.

This dynamic way of switching between multiple components can prove very useful cause we can use the same mount point for all of them and eliminates the need to find another way of having many components in one place, as we did in the previous examples.

15.1.2 Navigation

We can implement this way of swamping components to our known example of stories, where we used `*.vue` files to simulate a social network, where we had a login page, a registration page etc. Now we can use a page or tab system where we can put those previous components and each tab will contain one of them.

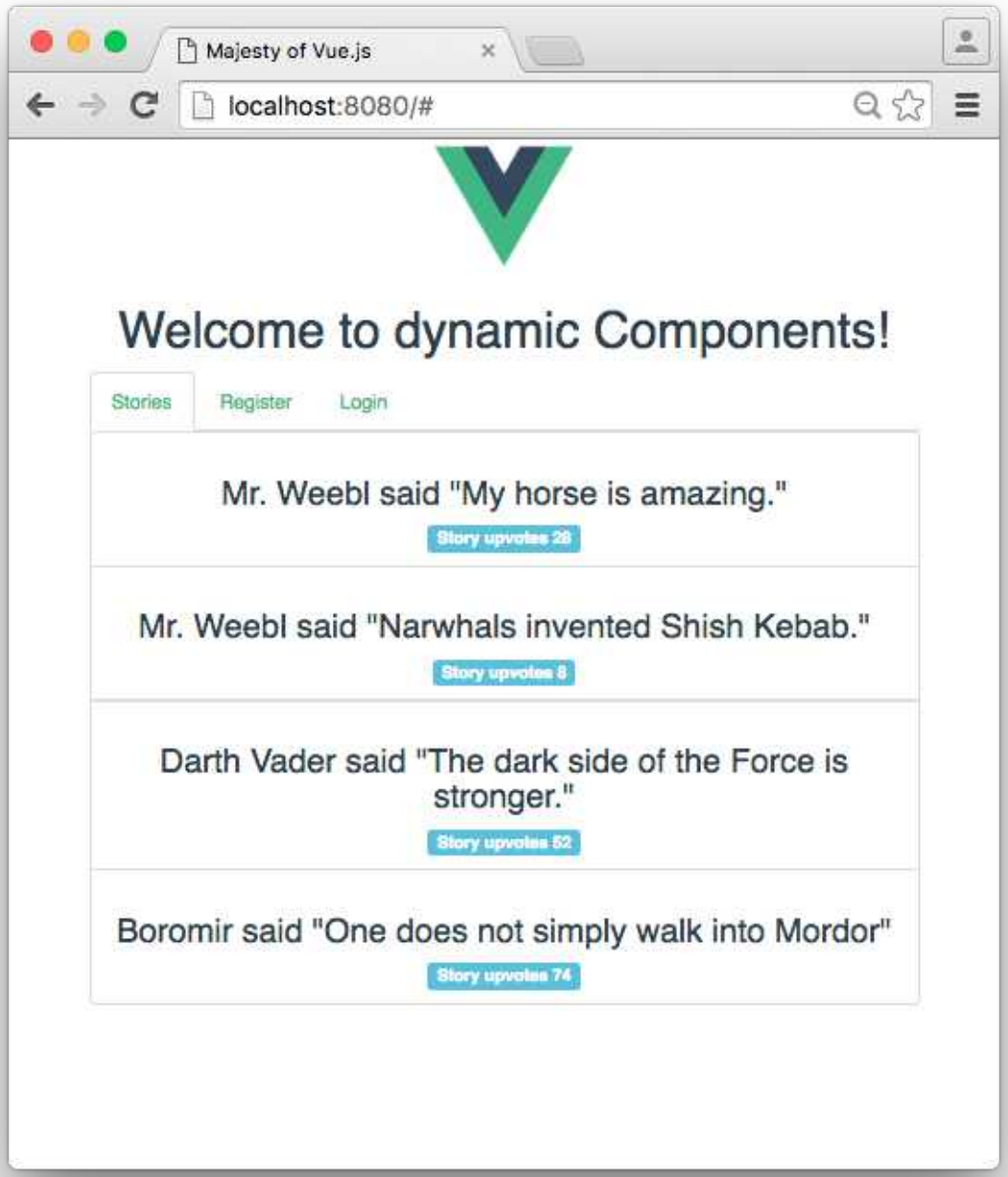
We are going to have `Stories.vue` in one tab, `Register.vue` in another, and `Login.vue` in a third. Don't forget that Register component has within it `Famous.vue`, which returns the most trending stories.

To achieve this, we will use tabs navigation to swap between them easy. Also we need a way of changing this tabs-components in a similar way as above. Read thoroughly the next example.

src/App.vue

```
<template>
  <div id="app">
    <img class="logo" src="./assets/logo.png">
    <h1>Welcome to dynamic Components!</h1>
    <ul class="nav nav-tabs">
    <!-- set 'active' class conditionally -->
      <li v-for="page in pages" class="{{isActivePage(page) ? 'active' : ''}}">
      <!-- use links to change between tabs -->
        <a @click="setPage(page)">{{page | capitalize}}</a>
      </li>
    </ul>
    <component :is="activePage"></component>
  </div>
</template>

<script>
// import Components as usual
import Stories from './components/Stories'
import Register from './components/Register'
import Login from './components/Login'

export default {
  components: {
    Stories,
    Register,
    Login
  },
  data () {
    return {
            // the pages we want to render each time
      pages: [
        'stories',
        'register',
        'login'
      ],
      activePage: 'stories'
    }
  },
  methods: {
```

```
40      setPage (newPage) {
41        this.activePage = newPage
42      },
43      isActivePage (page) {
44        return this.activePage === page
45      }
46    }
47  }
48  </script>
```



Pages for all components

Lets break it down.

We have introduced a "tab" system for each view we require. An array named `pages` contains the pages we would like to display and we are using `v-for` directive to create one tab for each of

them. To change between tabs we created a method called **`setPage`**, which changes the **`activePage`** data property on click, initially set to stories, to the clicked page. To determine which tab must be active an in-line **`if`** is applied, which sets the class **`active`** whether the current **`activePage`** property matches the page. Clearly we are importing each component as before, using **`<component :is="activePage"></component>`** again binded to a data property.

All other components remain the same as the previous examples, only this has changed to make tabs work.

With these few and simple lines of code, we have accomplished a simple navigation system, swapping between our components.

Code Examples

You can find the code examples of this chapter on GitHub[1].

[1] https://github.com/hootlex/the-majesty-of-vuejs/tree/master/examples/15.%20Swapping%20Components

16. Vue Router

Routing, in general, refers to determining how your application responds to a client request. A web browser request wouldn't be directed to your application without some form of routing. Router helps a web server to fetch an appropriate and exact information for the user. It is like a station master at a railway station, who informs the train operator when to change tracks.

The way of swapping views we just studied, paves the way for routing. The official router for Vue.js is called **vue-router**. It is deeply integrated with Vue.js core to make building Single Page Applications a breeze. This plugin is relatively easy to understand, install, and use.

The main features are:

- Nested route/view mapping
- Modular, component-based router configuration
- Route params, query, wildcards
- View transition effects powered by Vue.js' transition system
- Fine-grained navigation control
- Links with automatic active CSS classes
- HTML5 history mode or hash mode, with auto-fallback in IE9
- Restore scroll position when going back in history mode

These are just some of the provided features, you can see more in its Github pages[1].

Info

Here is the official vue-router library[2] and the documentation[3].

16.1 Installation

There are the usual ways to install the plugin; using cdn, NPM, and Bower. We are going to use the terminal to install it via NPM.

```
npm install vue-router
```

[1] https://github.com/vuejs/vue-router
[2] https://github.com/vuejs/vue-router
[3] http://router.vuejs.org/en/index.html

Type this command in your terminal to have it installed in your Node Modules folder inside your project's directory. After it is complete, go to your **main.js** file and add the following lines.

src/main.js

```
import VueRouter from 'vue-router'

Vue.use(VueRouter)
```

You can install a Vue.js plugin using **Vue.use()** as shown. For more information about **Vue.use** check the guide[4].

16.2 Usage

We need to create a router instance, where shortly after, we will pass extra options.

```
var router = new VueRouter()
```

We also need to define some routes. Each route should map to a component, which means that we are going to create routes for the ***.vue** files we used all along in our examples.

The main method to define route mappings for the router is **router.map(routeMap)**

src/main.js

```
router.map({
    '/': {
        component: require('./components/Hello.vue')
    },
    '/login': {
        component: require('./components/Login.vue')
    }
})
```

Inside the method we have defined 2 routes.

1. When `http://localhost:3000/` is met (or any port you might have), the default **Hello.vue** will be rendered
2. When `http://localhost:3000/login` is met, the **Login.vue** will be rendered.

[4] http://vuejs.org/api/#Vue-use

Info

We are using **`require()`** because these components are not imported in this file. Alternatively you can import those you need and then do **`'/foo': {component: Foo }`**

We need an outlet for the router. The guide states that the router needs a root component to render. We will use the **`App.vue`** component as root.

Start the router-enabled app with **`router.start()`**

src/main.js

```
1  // previous code
2
3  router.start(App, 'body')
```

The router will create an instance of App and mount to the element matching the selector, here the **`body`**. Since we are already importing **`App`** in our **`main.js`** file, we are set here.

Now head to **`App.vue`**, to make the changes required.

src/App.vue

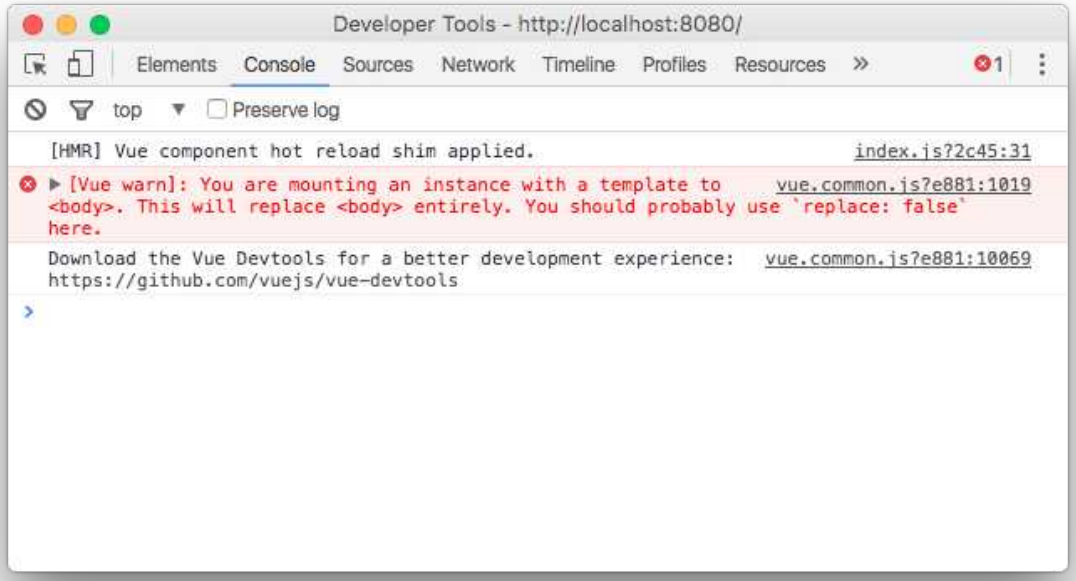
```
1  <template>
2    <div id="app">
3      <img class="logo" src="./assets/logo.png">
4      <h1>Welcome to Routing!</h1>
5      <a v-link="{ path: '/' }">Home</a>
6      <a v-link="{ path: '/login' }">Login</a>
7      <!-- route outlet -->
8      <router-view></router-view>
9    </div>
10 </template>
```

The **`<router-view></router-view>`** is the place where all the magic happens while components are being rendered for us.

`v-link` is the directive for enabling user navigation in a router-enabled app. We can simply state a **path** to match a route we defined in **`main.js`**. **`v-link`** can take more arguments for more complex navigations, which we will review soon.

Warning

After the page has loaded we get a warning from Vue: `[Vue warn]: You are mounting an instance with a template to <body>. This will replace <body> entirely. You should probably use 'replace: false' here.`

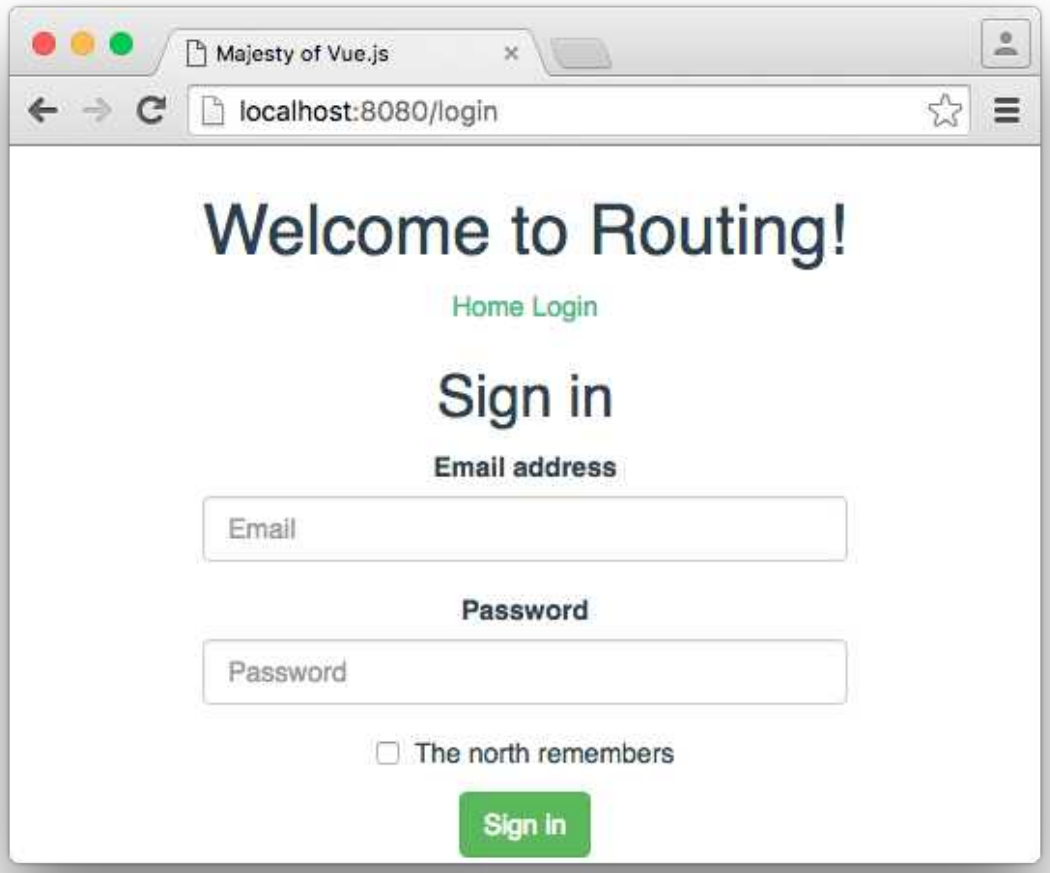


Vue warning

As good followers of Vue, we submit to its will.

src/App.vue

```
export default {
  replace: false,
}
```



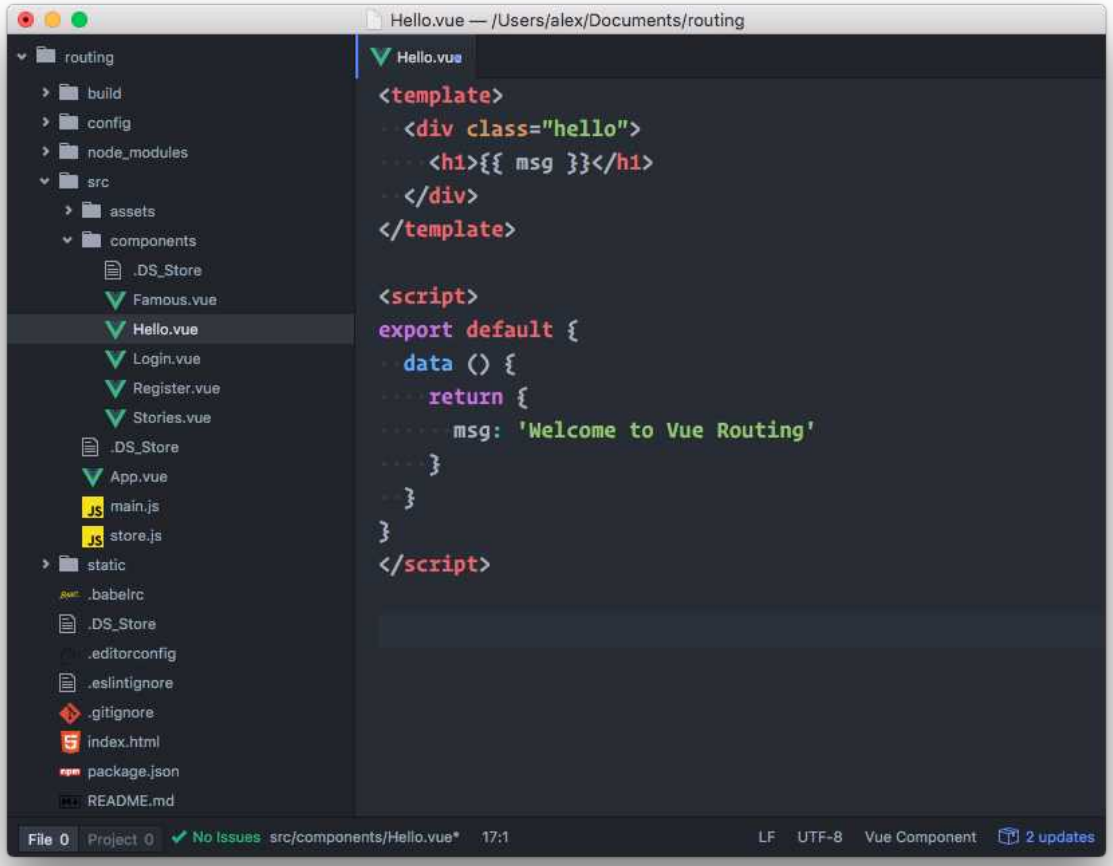
Login route

Everything seems to run smoothly here, time to move on.

16.3 Nested routes

Nested routes, are routes that live within other routes. Mapping nested routes to nested components is a common need, and it's also very simple with vue-router.

All we have to do is to declare "subRoutes" and use a nested `<router-view>`.



Project structure

Assuming we have the above structure, we can proceed to the following changes.

src/main.js

```
router.map({
    '/': {
        component: require('./components/Hello.vue')
    },
    '/login': {
        component: require('./components/Login.vue')
    },
    '/stories': {
        component: require('./components/Stories.vue'),
         subRoutes: {
            '/famous': {
                component: require('./components/Famous.vue')
```

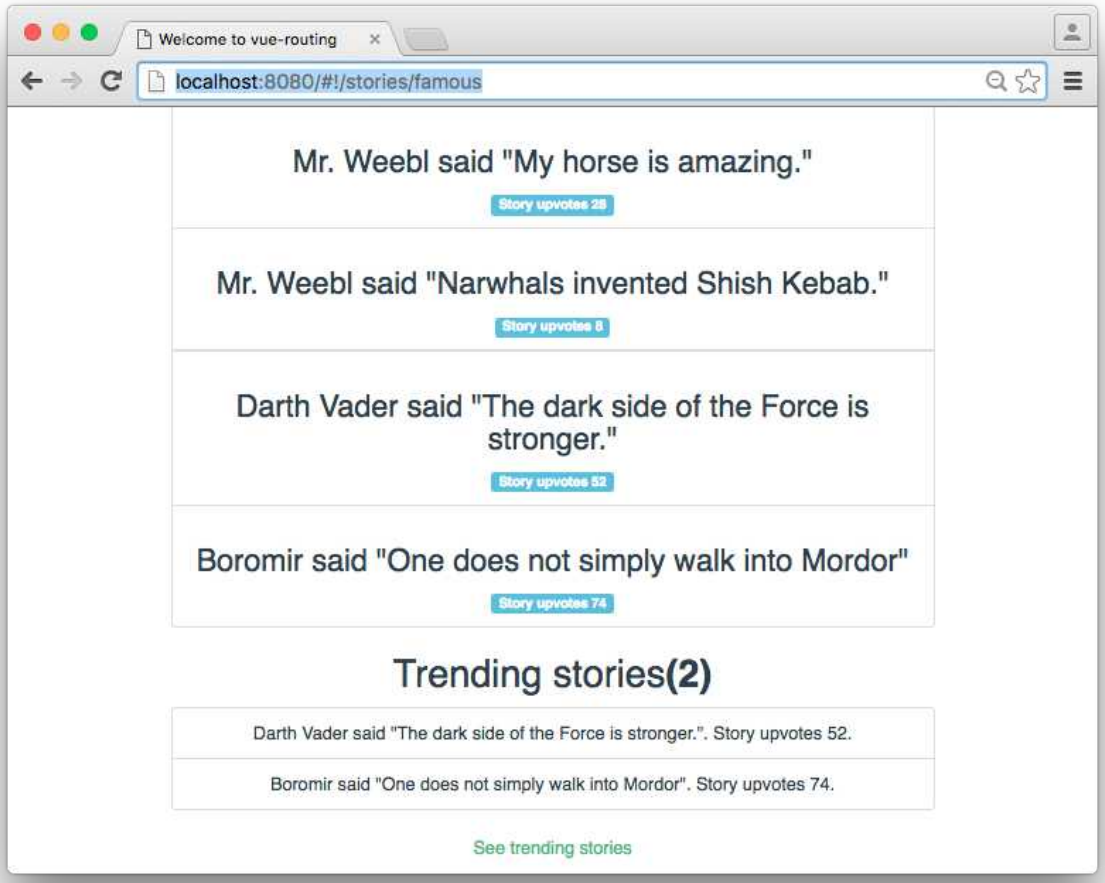
```
            },
        }
    },
})
```

As seen above, we've created a new route linked to **`Stories.vue`** and a subRoute within, linked to **`Famous.vue`**.

src/Stories.vue

```
<template>
        ...
    // nested outlet
    <router-view></router-view>
    <a v-link="{ path: '/stories/famous' }">Trending stories</a>
</template>
```

When **`/stories`** is matched, only the contents of **`Stories.vue`** component will be rendered. Famous will be rendered inside Stories' **`<router-view></router-view>`** when **`/stories/famous`** is matched.



SubRoute

16.4 Route Matching

While the options we reviewed for routing serve our needs in a small project like our example, presumably you will need more options as your project grows. For instance, if we decide later to change '/login' url to '/signin', we will have to update all the links directing to the login page. To prevent this from happening, we can give each route a name.

16.4.1 Named Routes

To name a route we have to update the route configuration file.

src/main.js

```
router.map({
    '/': {
        name: 'home', // give the route a name
        component: require('./components/Hello.vue')
    },
    '/login': {
        name: 'login', // give the route a name
        component: require('./components/Login.vue')
    },
    '/stories': {
        name: 'stories',
        component: require('./components/Stories.vue'),
         subRoutes: {
            '/famous': {
                component: require('./components/Famous.vue')
            },
        }
    },
})
```

We can give a name to a route by adding a **name** property and use our own identifier to link it afterwards.

src/App.vue

```
<template>
  <div id="app">
    <img class="logo" src="./assets/logo.png">
    <h1>Welcome to Routing!</h1>
    <!-- instead of 'path' use name: 'yourName' -->
    <a v-link="{ name: 'home'  }">Home</a>
    <a v-link="{ name: 'login' }">Login</a>
    <!-- route outlet -->
    <router-view></router-view>
  </div>
</template>
```

Changing from **path** to **name** here is not necessary, but it can prove useful.

Before moving on, I would like to point out something regarding the browser URL, when using vue-router. As you have seen, when a route changes, a '#!' is appended to the URL. For instance, the URL is **/#!/login**, when we navigate to the login page.

We are going to fix this using the option **`history`** instead of **`hashbang`** because setting the second to `false` only hides the exclamation mark. Also we will set another option, **`root`**, which defines a root path for all router navigations. Changing this to anything from its initial value, that equals `null`, will result in paths which will always include the new value in the actual browser URL.

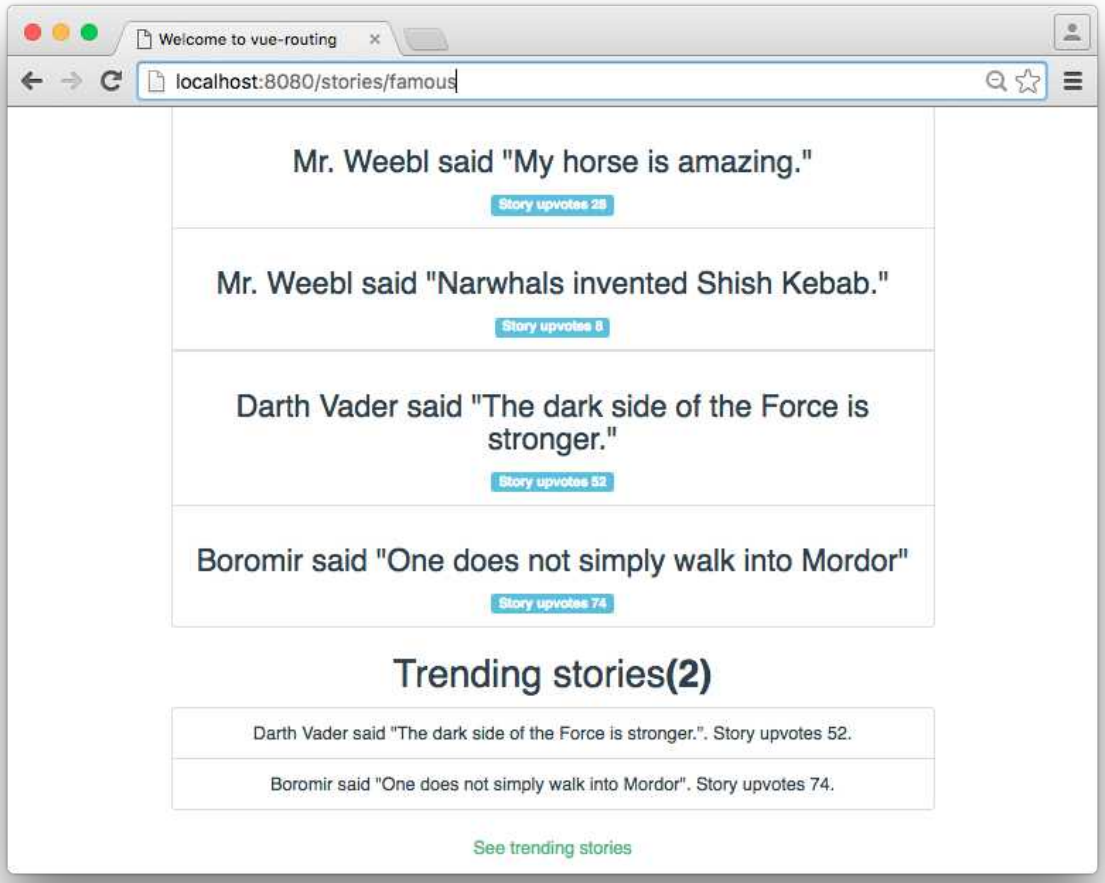
For example if we set here **`root`** to **`/vuejs`** the login page will be **`/vuejs/login`**.

You can set anything to be the **`root`**, here let's just put **`/`**.

src/main.js

```
var router = new VueRouter({
        history: true,
        root: '/'
})
```

This relieves us from **`#!`** in the URLs.



Neat URLs

Info

Check the detailed list of available options on vue-router's documentation[5]

16.4.2 Route Object

All information of a parsed route will be accessible on the exposed **Route Context Objects**, also called "**route" objects**.

The route object will be injected into every component in a vue-router-enabled app as **this.$route**, and will be updated whenever a route transition is performed.

Below is a list with `$route` object's properties.

[5] http://router.vuejs.org/en/options.html

Property	Description
path	A string that equals the path of the current route, always resolved as an absolute path. e.g. "/foo/bar".
params	An object that contains key/value pairs of dynamic segments and star segments. More details below.
query	An object that contains key/value pairs of the query string. For example, for a path /foo?user=1, we get $route.query.user == 1.
router	The router instance that is managing this route (and its owner component).
matched	An array containing the route configuration objects for all matched segments in the current route.
name	The name of the current route, if it has one.

16.4.3 Dynamic Segments

Vue Router provides the ability to form paths using dynamic segments. Dynamic segments are segments with a leading colon. They are called dynamic because their value is changeable.

Info

URL segments are the parts of a URL or path delimited by slashes. If you had the path **`/user/:id/posts`**, then "user", ":id", and "posts" would each be a segment.

In this path, the **`:id`** is the dynamic segment and will match **`/user/11/posts`**, **`/user/37/posts`**, etc.

When a path containing a dynamic segment is matched, the dynamic segments will be available inside **`$route.params`**.

In our example, we can use a dynamic segment to access a certain story by its **`id`**, in order to create a view where we can edit it.

src/main.js

```
router.map({
    '/': {
        name: 'home',
        component: require('./components/Hello.vue')
    },
    '/login': {
        name: 'login',
        component: require('./components/Login.vue')
    },
    '/stories': {
```

```
        name: 'stories',
        component: require('./components/Stories.vue'),
         subRoutes: {
            '/famous': {
                component: require('./components/Famous.vue')
            },
        }
    },
    '/stories/:storyId/edit': {
        name: 'edit',
        component: require('./components/Edit.vue'),
    },
})
```

Now, we need a way to link **`Stories.vue`** with **`Edit.vue`**. Let's manipulate the file.

Note

I have created **`Edit.vue`** file in the background. You will see it soon, after the routing for it is complete.

src/component/Stories.vue

```
<template>
  <ul class="list-group">
    <li v-for="story in stories"  class="list-group-item">
      <h3>{{ story.writer }} said "{{ story.plot }}"</h3>
      <button class="btn btn-default" v-link="{ name: 'edit'}">Edit</button>
      <span class="label label-info">Story upvotes {{ story.upvotes }} </span>
    </li>
  </ul>
</template>

<script>
import {store} from '../store.js'

export default {
  data () {
    return {
      stories: store.stories,
      local: 'data'
    }
  }
```

```
}
</script>
```

Note

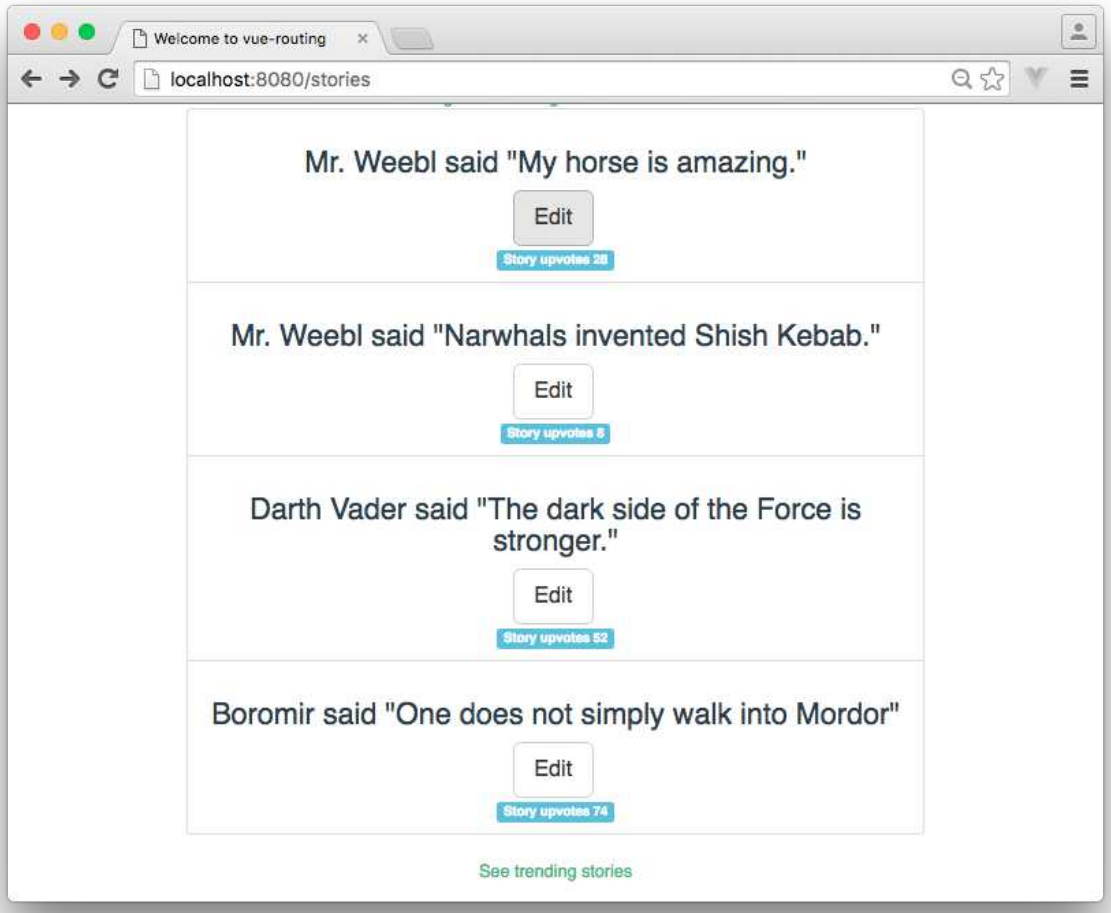
Our example files are almost the same as before, with minor alterations, mostly styles, which do not affect their functionality. A noteworthy change is the addition of an **`id`** to each story within **`store.js`**.

We added a button to link the `edit` route. Well, this is not enough, because we also need to pass the story's **`id`** to the 'edit' route.

To do so, we are going to edit **`v-link`** directive and make the **`:storyId`** turn into the corresponding `id` of each story.

src/component/Stories.vue

```
<template>
  <ul class="list-group">
    <li v-for="story in stories"  class="list-group-item">
      <h3>{{ story.writer }} said "{{ story.plot }}"</h3>
      <button class="btn btn-default"
        v-link="{ name: 'edit', params: { storyId: story.id }}"
      >
        Edit
      </button>
      <span class="label label-info">Story upvotes {{ story.upvotes }}
      </span>
    </li>
  </ul>
</template>
```



Ready to edit a story

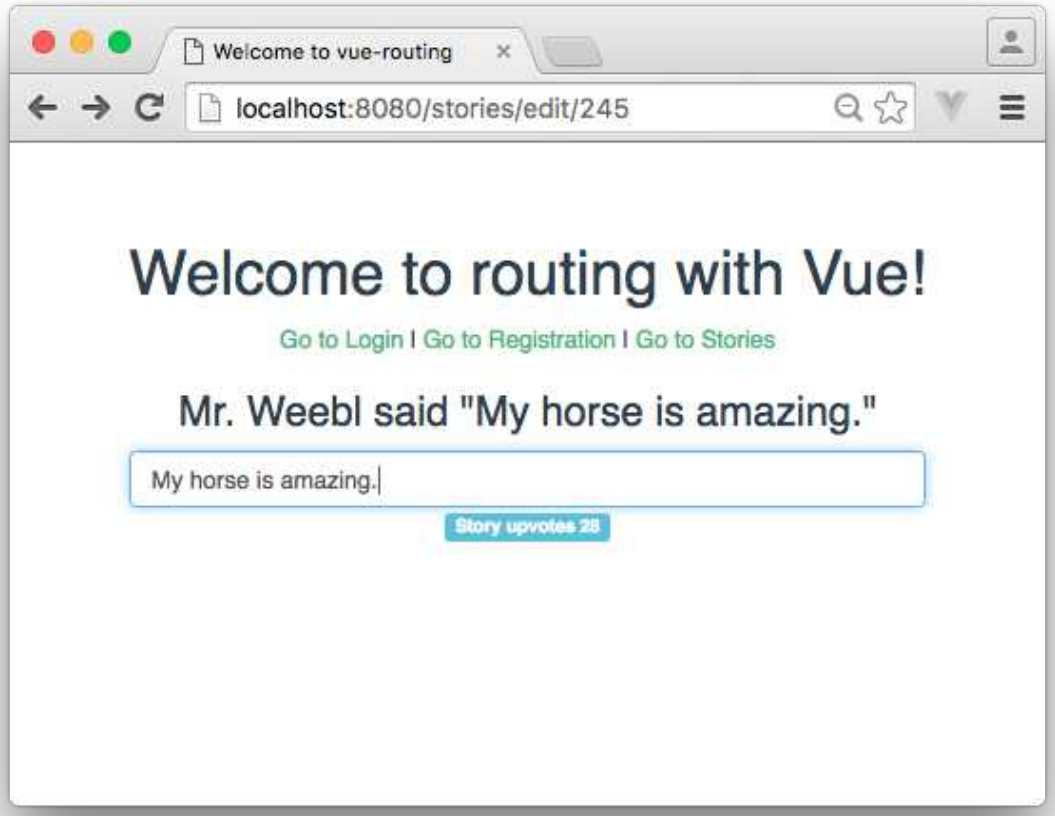
Within **`$route.params`**, the **`id`** of the chosen story is available when we reach our destination. With this at hand, we can pick out the story that the user wants to edit, and bring it to him.

It's time to show the **`Edit.vue`** file, where the editing will take place.

src/components/Edit.vue

```
<template>
  <div class="form-horizontal">
    <h3>{{ story.writer }} said "{{ story.plot }}"</h3>
    <input type="text" class="form-control" v-model="story.plot">
    <span class="label label-info">Story upvotes {{ story.upvotes }} </span>
  </div>
</template>

```

```
<script>
import {store} from '../store.js'

export default {
  ready () {
    this.story = store.stories.find(this.isTheOne)
  },
  data () {
    return {
      story: {}
    }
  },
  methods: {
    isTheOne (story) {
      return story.id === this.$route.params.storyId;
    },
  }
}
</script>
```



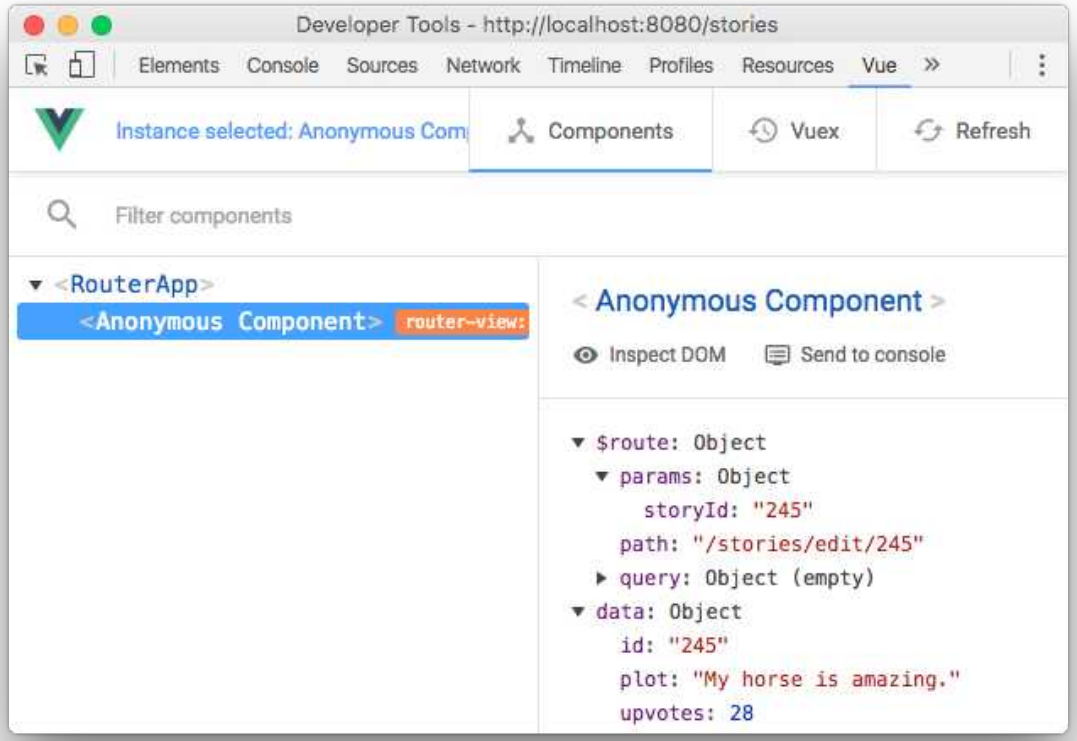
Editing selected story

Here the story chosen is ready for editing.

We are using story's id from **`$route.params`** object, to pick the desired `story` out of `stories` array with JavaScript's find method[6].

Notice that the `id` of the story is shown in the URL.

[6] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find



Inside $route

16.4.4 Route Alias

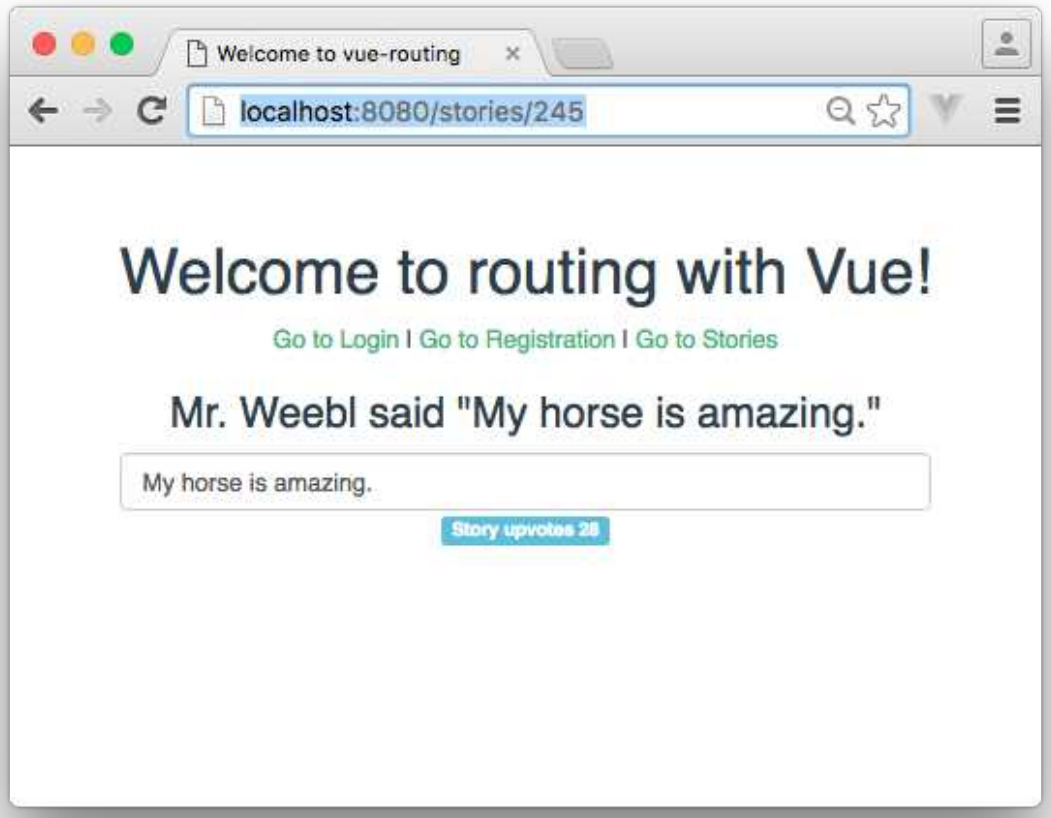
When we define new routes we usually try to make them clear and representative. Sometimes though, we might end up with long paths or complex ones, which can be difficult to handle later.

When we want to directly edit a story through the browser URL, we have to visit `'/stories/:storyId/edit'`. We can make this easier by defining a global alias for this route, where a shorter URL would lead us to the same place.

src/main.js

```
router.alias({
  // match '/stories/:storyId' as if it is '/stories/:storyId/edit'
  '/stories/:storyId': '/stories/:storyId/edit'
})
```

Using this configuration we can just type the left part, the alias, and it will match as if the right is typed.



Using route alias

16.4.5 Route Go

At some point, we want to navigate to a route not through links, but programmatically.

To navigate to a route, we can use `router.go(path)`. The path can be either a string or an object.

If it is a string, the path must be in the form of plain path, meaning that it can not contain dynamic segments. For example `router.go('/stories/11/edit')`.

If it is an object it can be in two forms:

```
{ path: '...' }
// ex: router.go({ path: '/stories/11/edit' })

// ~~OR~~

{
  name: '...',
  // params and query are optional
  params: { ... },
  query: { ... }
}
// ex: router.go({ name: 'edit', params: {storyId: '11'} })
```

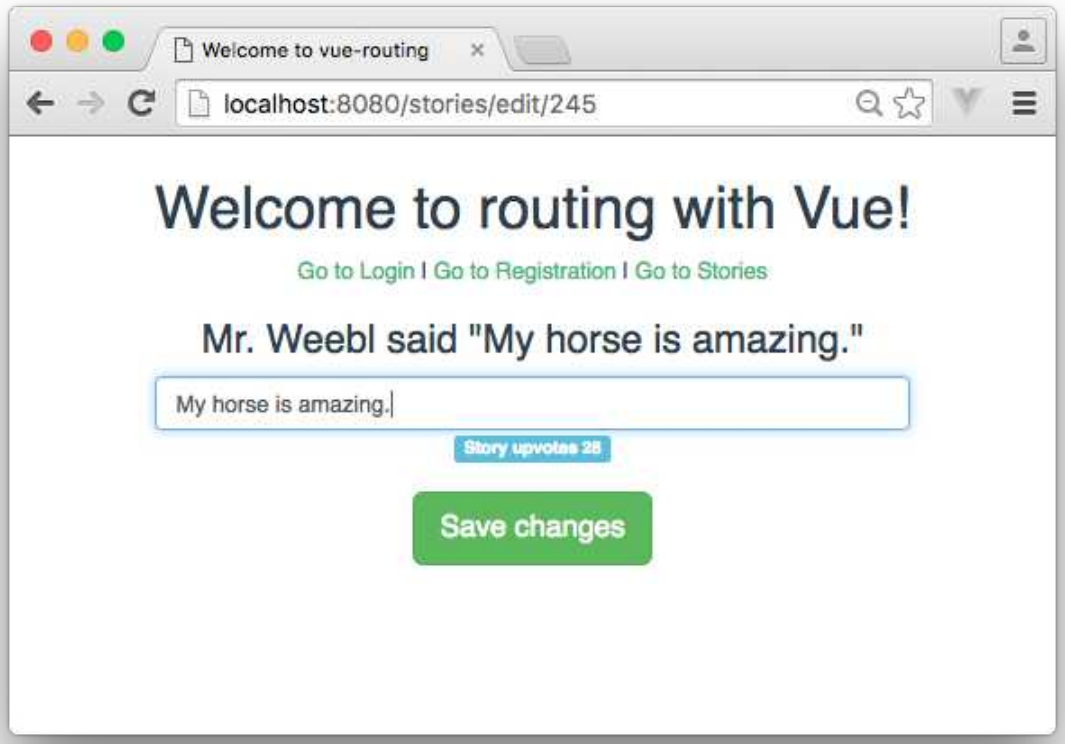
We can use **router.go** to navigate to the stories' listing page after editing is complete.

src/components/Edit.vue

```
<template>
  <div class="form-horizontal">
    <h3>{{ story.writer }} said "{{ story.plot }}"</h3>
    <input type="text" class="form-control" v-model="story.plot">
    <span class="label label-info">Story upvotes {{ story.upvotes }} </span>
  </div>
  <button @click="saveChanges(story)" class="btn btn-success btn-lg">Save change\
s</button>
</template>

<script>
export default {
  ready () {
    this.story =  store.stories.find(this.isTheOne)
  },
  data () {
    return {
      story: {}
    }
  },
  methods: {
    saveChanges (story) {
      console.log('Saved!')
      this.$router.go('/stories')
    },
    isTheOne (story) {
```

```
27        return story.id === this.$route.params.storyId;
28      },
29    }
30  }
31  </script>
```

We have created a new method named **saveChanges**. When called, it logs a message to the console (for simplicity) and using **this.$router.go()**, navigates back to **Stories.vue**.



Completed editing file

16.4.6 Filtering Transitions

Vue Router provides a convenient mechanism for filtering transitions. To filter a transition you can use **router.beforeEach()** which is triggered before each transition, and **router.afterEach()** which is triggered after.

router.beforeEach() can be handy in a scenario regarding authorization. For example if a user does not have permission to access a page of your app, he should be directed to the login page.

Let's see how can we can accomplish that in a short example.

src/main.js

```
// create a dummy user object
var User = {
  isAdmin: false
}

router.beforeEach(function (transition) {
  if (transition.to.path != '/login' && !User.isAdmin) {
    // if not going to login and not an admin redirect to login
    router.go('/login')
  } else {
    //if authorized proceed
    transition.next()
  }
})
```

Here we apply a rule so router won't let users proceed to any page except login.

Code Examples

You can find the code examples of this chapter on GitHub[7].

[7] https://github.com/hootlex/the-majesty-of-vuejs/tree/master/examples/16.%20Vue%20router

16.4.7 Homework

Throughout this chapter we analyzed a lot of things and it's been a while since we've assigned you some homework! To complete this one, you will probably need to review what he have already said .

The app you have to build is a mini Pokédex.

The home page will show a list of Pokémon categories, such as *Fire, Water, etc.* From there, the user will be able to browse a category, view its Pokémon, and add new ones.

Your routes could be something like these:

Route	Description
/	List categories.
/category/:name	Show category's Pokémons.
/category/:name/pokemons/new	Add new Pokémon to category.

Each transition must be logged to the console. For example, when the user decides to browse category *Fire*, a message has to be logged, informing the user that he is going to visit /category/Fire.

We have created the Pokédex object to help you get started. You can find it here[8].

Info

The `/category/:name/pokemons/new` route is a subroute of `/category/:name`.

When the user visits `/category/:name/pokemons/new` s/he should see a form to add a new Pokémon along with the listing of the Pokemon's category.

Hint 1

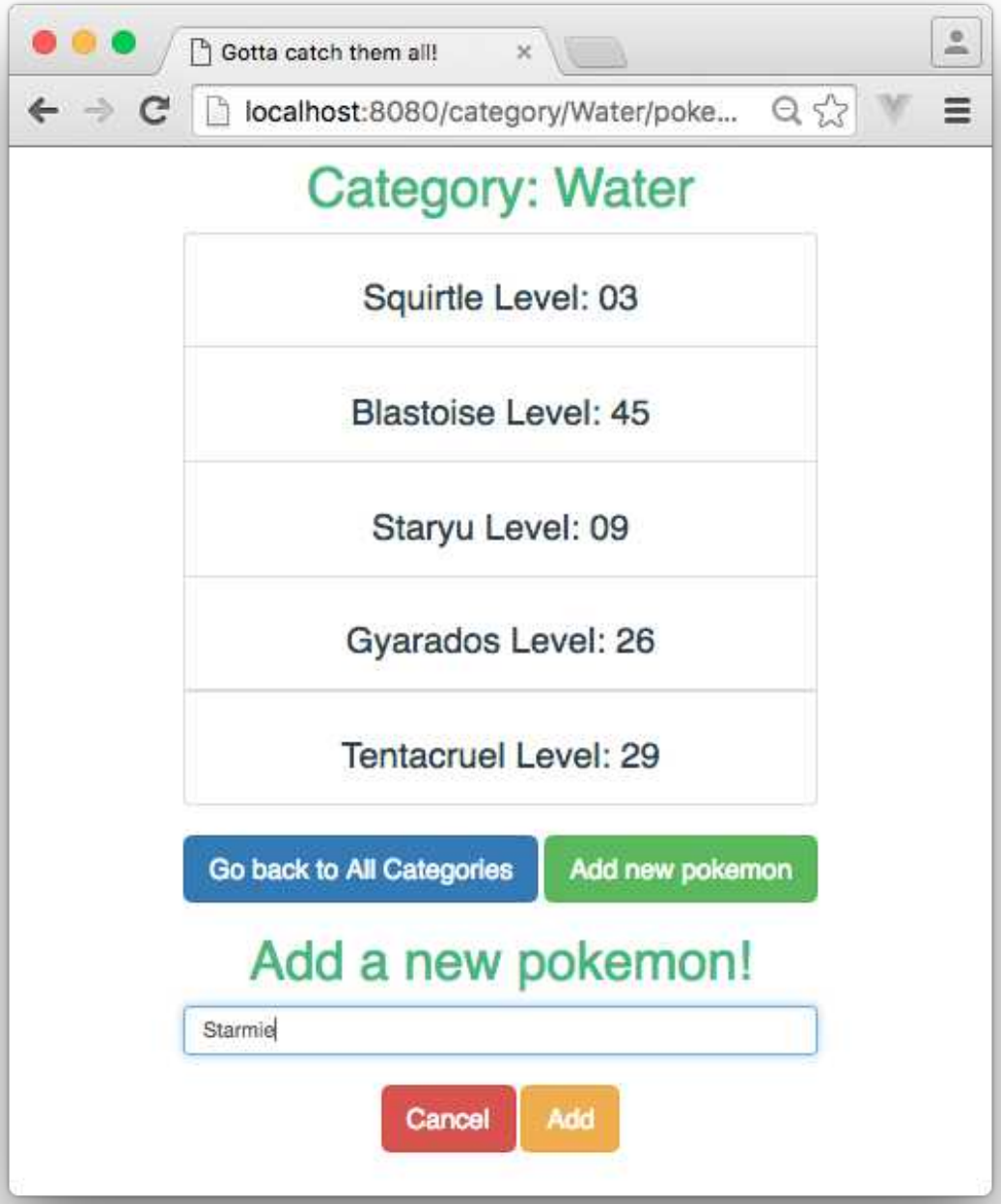
To access Pokémon of a specific category, consider using JavaScript's find method[9].

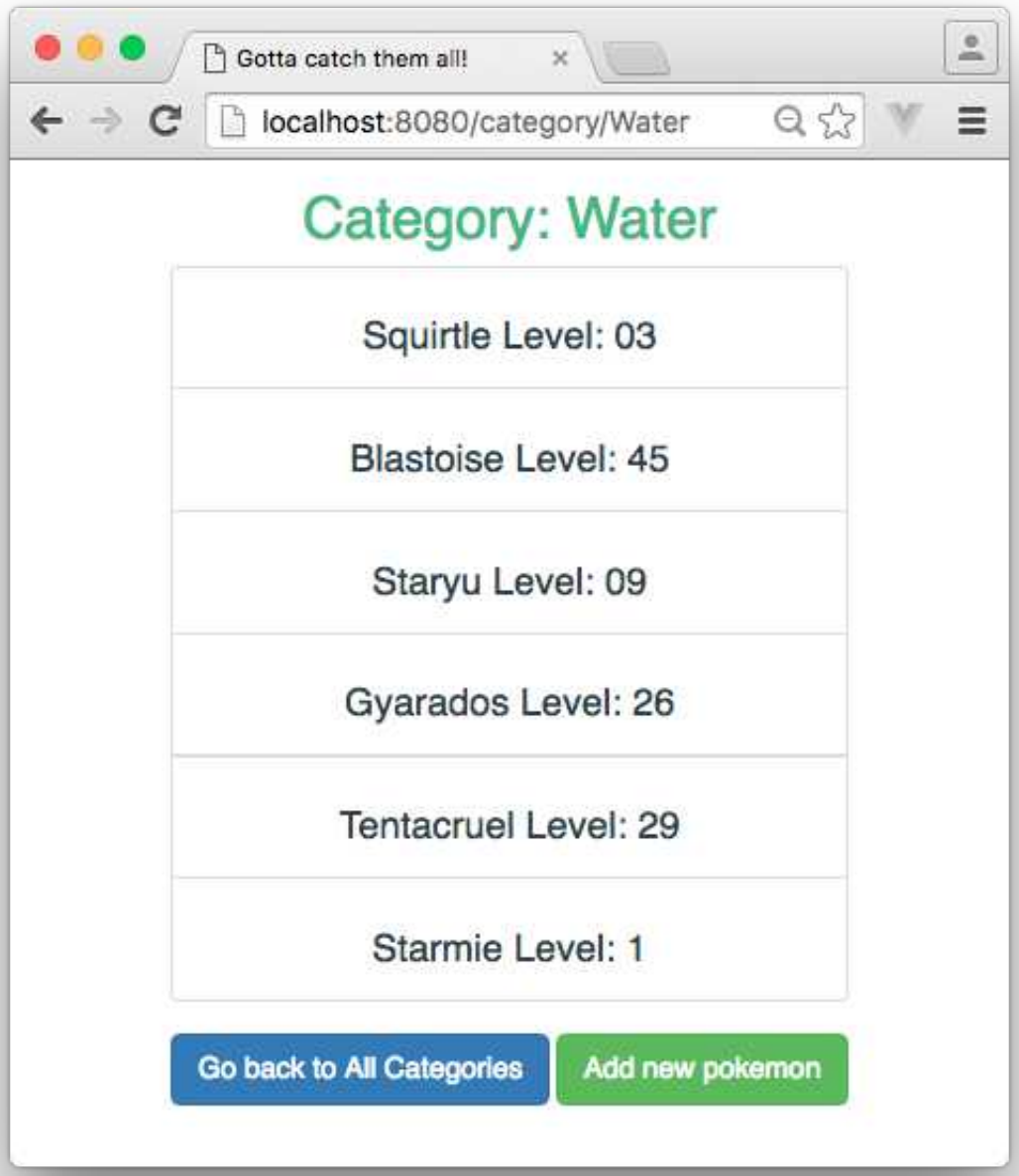
Hint 2

To log messages to the console before each transition use `router.beforeEach()`.

You access the target route within `transition.next()`

[8] https://github.com/hootlex/the-majesty-of-vuejs/blob/master/homework/chapter16/src/pokedex.js
[9] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find


Gotta catch them all!
localhost:8080/category/Water/poke...
Category: Water
Squirtle Level: 03
Blastoise Level: 45
Staryu Level: 09
Gyarados Level: 26
Tentacruel Level: 29
Go back to All Categories
Add new pokemon
Add a new pokemon!
Starmie
Cancel
Add



You can find a potential solution to this exercise here[10]

[10] https://github.com/hootlex/the-majesty-of-vuejs/tree/master/homework/chapter16

Closing Thoughts

Well, it seems that our journey has come to an end. We hope that you have enjoyed reading the book as much we enjoyed writing it. We also hope that you liked (at least a bit) our sense of humor.

We want to thank each one of you, for reading the entire book and of course for supporting our efforts to make this book a reality. Huge thanks to all the contributors[11], who spared some of their time to report several grammatical, syntax, and coding flows.

I know, you are all looking forward to the next edition, which will demonstrate how to build all these things using Vue.js 2. Don't worry, we'll do our best to have it published as soon as possible.

In case you missed it, we have launched Vue.js Feed[12], where news, tutorials, code, and plugins, related to Vue.js, are published **daily**. We'd love it if you would like to be part of this, since we love you all. All and every single one of vue. :)

Don't be a stranger!

For any question/clarification don't hesitate to contact us on Twitter[13]. We'll also keep you updated and inform you about the new edition.

---

[11] https://github.com/hootlex/the-majesty-of-vuejs

[12] https://vuejsfeed.com/

[13] http://twitter.com/tmvuejs

Further Learning

We have curated some tutorials and books that you should consider reading. We've also gathered a few open source projects for you to see Vue in real action. Some of the resources are not directly related to Vue.js but they refer to related subjects.

Tutorials

- Peeking into Vue.js 2[14] - All these new features that Vue 2 comes with, sound great. Though, before you use them, you should get a bit familiar with what exactly each one is.
- Form Validation using Vue.js 2[15] - Learn about some of the most popular Vue.js form validation plugins and how to implement them in your project.
- How to build a reactive engine in JavaScript.[16] - This tutorial goes the getters/setters way of observing and reacting to changes in real time.
- Building a Mobile App with Cordova and Vue.js[17] - This tutorial will show you how to create a simple mobile app, which generates random words by using Cordova and Vue.js.
- Simple guide to authoring open-source Vue.js components[18] - Here are some bits of advice, for those that own or think about starting their own open source solutions for Vue.js.
- Best Free Vue.js Learning Resources[19] - This post offers a lot of resources you'll need to go from a Vue.js newbie to building your first (of many) applications.

Videos

- Learning Vue 1.0: Step By Step[20] - A series of lessons demonstrating the building blocks of Vue.
- Demystifying Frontend Framework Performance[21] - In this talk, Evan You will walk you through the techniques used in major front-end frameworks - namely dirty checking, virtual-dom diffing and dependency-tracking.

[14] https://dotdev.co/peeking-into-vue-js-2-part-1-b457e60c88c6
[15] https://dotdev.co/form-validation-using-vue-js-2-35abd6b18c5d
[16] http://monterail.com/blog/2016/how-to-build-a-reactive-engine-in-javascript-part-1-observable-objects/
[17] https://coligo.io/building-a-mobile-app-with-cordova-vuejs/
[18] http://monterail.com/blog/2016/simple-guide-to-authoring-open-source-vue-js-components/
[19] http://whatpixel.com/vuejs-learning-resources/
[20] https://laracasts.com/series/learning-vue-step-by-step
[21] https://vuejsfeed.com/blog/demystifying-frontend-framework-performance-video

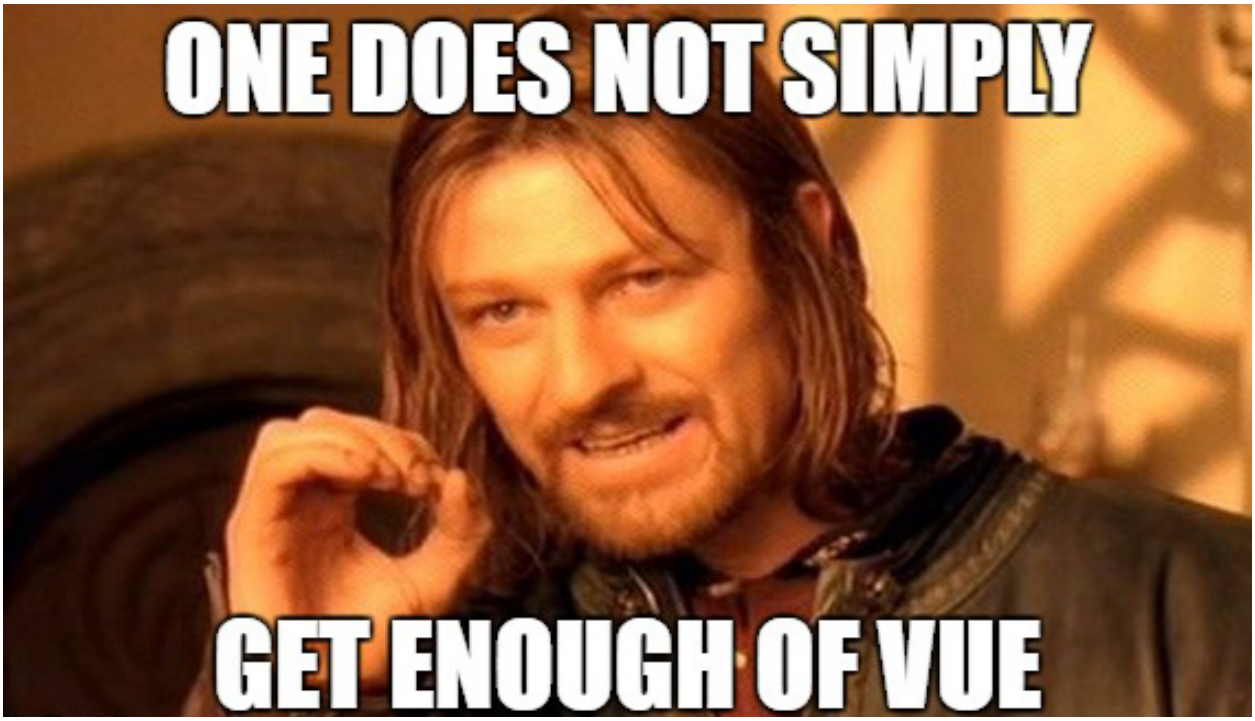
Books

- Understanding ECMAScript 6[22] - There's a lot of new concepts to learn and understand in ES6. Get a headstart with this book!
- Build APIs You Won't Hate[23] - These days it's pretty standard to build your application with a separation of the frontend and backend logic. Frontend is done pretty well in JavaScript in the browser using awesome frameworks like Vue.js, and the backends will usually be some server-side language knocking out JSON.

Open source projects

- Vuedo[24] - A blog platform, built with Laravel and Vue.js.
- ConFOMO[25] - A simple tool that makes it easy to track your friends at conferences.
- Ribbbon[26] - A project management system for artisans.
- Airflix[27] - An AirPlay friendly web interface to stream your movies and TV shows from a home server.
- Koel[28] - A personal music streaming server that works.

[22] https://leanpub.com/understandinges6
[23] https://leanpub.com/build-apis-you-wont-hate
[24] https://github.com/Vuedo/vuedo
[25] https://github.com/mattstauffer/confomo
[26] https://github.com/canvasowl/ribbbon
[27] https://github.com/wells/airflix
[28] https://github.com/phanan/koel



The End...