Table of Contents

Introduction

Welcome!

This book explains how to build a website using *Vue.js*, a progressive Javascript framework for building user interfaces. Vue is lightweight enough to be used in an existing website to introduce some extra functionality, or as the main front end framework for a professional grade, large scale SPA (single page application).

This books assumes basic knowledge of HTML, CSS and Javascript, but starts from scratch, so anyone with a programming background should be able to pick it up fairly easily. Unlike some others books, I want to also introduce and explain how the tooling works, which is relevant to developing any modern website, not just ones using Vue.

One of the great things about Vue is its simplicity. To get started, all you need to do is include it in a `<script>` tag at the top the main `index.html` file. However, when building large, complex apps, tools such as *yarn* and *npm* (package managers), *webpack* (a bundler), *mocha, chai _and _karma* (unit testing tools) and others. If you are not familiar with these, don't worry! I will introduce them incrementally and explain how and why they are used.

Without further ado, let's get started.

Chapter 3: A modern workflow with webpack

3.1 More modules! Yarn/NPM

So far, we have developed our applications in two files: `index.js` for all the Javascript code, mostly business logic, and `index.html`, where we build the UI. As an application grows in size and complexity, this becomes impossible to manager. Before the days of package managers, tools like webpack and browerify, and Node.js (don't worry if you dont know about any of those yet - you soon will), people would just include a bunch of <script> tags at the top of the page.

Fortunately, there is a better way now! Introducing **Node.js**. We will not be using Node.js itself - Node itself is a client side Javascript architecture, which is not the focus of this book - however, it comes with NPM (*node package manager)* which is used to retrieve, install and update modules for most Javascript based applications nowdays. After some setup, you can simply run `npm install [module]` and in your and `import [package] from 'package'` to use any of the thousands of modules available.

Next, **Yarn** - another package manager for Javascript modules, using the same registry as NPM. Yarn addresses some of the problems NPM taught us, and has some other nice features, so we will use Yarn for the rest of this book.

Before getting into too much more depth, let's see Yarn in action. Firstly, download Node.js and install it - it comes with NPM. To use Yarn, you need to use a *terminal*, also known as the *command prompt.* If you are on Windows, *Powershell* should be preinstalled, and *terminal* on macOS/Linux. Open your command line of choice.

3.2 Installing and running commands from the command line

If you have successfully installed Node, you should be able to run the following command:

```
npm --version
```

and it shoud print something like `4.0.5`. If not, troubleshoot until you are able to get this to work.

Next, we will install Yarn using NPM! Installing a package manager with another package manager? Think installing Google Chrome using Internet Explorer. Run:

```
npm install --g yarn
```

Where -g means 'global' - so you can use Yarn from any directory on your PC. When Yarn finishes installing, we are ready to get started... after a quick crash course using the terminal. Some simple commands that will work in macOS's terminal and Windows Powershell:

- `ls`

This command will show the contents of the current directory in the terminal window.

- `cd` *(stands for change directory)*

Used to change directories from within a terminal. `cd ..` will take you up one level, and `cd [folder]` will take you into the folder. Try moving around from wherever your terminal lands you by default to get used to it. If you get lost, just use `ls` to see all the files in the current directory, and `pwd` on macOS or `dir` on Windows to see the current location.

- `mkdir [directory name]` on macOS/Linux will create a new folder called [directory name]. Give it a try a few times
- `New-Item [directory name]` is the Windows equivalent.
- `touch [file name]` will create a new file called [file name] on macOS.
- `New-Item [file name]` is the Windows equivalent.

This should be enough to get started. Using `cd`, navigate to the directory you want to build the following application (not this chapter's app, but a simple example). On macOS, I type `cd` without any arguments to get back to my root directory. Then I type `mkdir yarn-test-app` to create new folder for my test app, and then `cd yarn-test-app` to enter the folder.

3.3 Modules! Managing your first app with Yarn

Once you've gotten to the location you want to make your application, create a new directory using the relevant command, and `cd` into that directory. Then run `yarn init -y` to get started! Typing `ls` should now display the contents of the folder, which contains a single file called `package.json`. This will hold all the information needed to get our application up and running.

Next, create a file called `index.js` using the command line. On macOS, I simply type `touch index.js`. This will be the starting point for a simple test app, which will simply greet the user (from the command line). Let's install a package to help us do that: **greeting**. Run:

```
yarn add greeting
```

And you should see a bunch of words in the terminal, and a success message if everything goes well. Let's use **greeting** to say hi. Inside of `index.js`, enter the following:

```
const greeting = require('greeting')

let randomGreeting = greeting.random()
console.log(randomGreeting)
```

To use the **greeting** package, simply `require` it! To run this program, type `node index.js`. Node, the server side Javascript architecture, will run the script and output a random greeting. Trying running it again for some other greetings.

Notice `package.json` now looks like this:

```
// index.js
...
"dependencies": {
  "greeting": "^1.0.6"
}
...
```

So if someone else was to run `yarn install`, the required packages would be installed, based on the `package.json` file.

3.4 Bundle it up: Webpack

Webpack is a tool to build your Javascript applications. Modern Javascript projects consist of many modules, which have to be included in the correct order to operate correctly. Webpack does all this, and more, for us. By setting up webpack, with a single command, we can compile our entire application, including all the components, and modules, into a single .js file, which is then included in our HTML.

First, add webpack using yarn:

`yarn add webpack`

Great! We need to also create the file that webpack will write our bundled application too. In the root directory, create `bundle.js` by hand or by running `touch bundle.js`.

Webpack uses a different system for requiring modules - the ideas are the same, however where Node uses `require`, webpack uses `import`. Modify `index.js` as follows:

```
import greeting from 'greeting'

const sayHi = (message) => {
  let el = document.createElement('div')
  el.innerHTML = messagae
  return el
}

document.body.appendChild(sayHi(greeting.random())
```

`sayHi` creates a div with a random greeting, which is appended to the body of the webpage. Go ahead and make the webpage called `index.html` and insert the following:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title></title>
  </head>
  <body>
    <script src="bundle.js"></script>
  </body>
</html>
```

To compile the application, run `webpack index.js bundle.js`. Opening `index.html` in your browser should show a random greeting, that changes every time you refresh! Take a look in `bundle.js`, and you will see not only the `sayHi` function, but the source for the greeting module as well.

Webpack does a *lot* of things. This is a simple example, and to save time and effort, the projects from now on will use premade webpack templates from the Vue community, that include all the modules required, a test setup, and more.

Chapter 4: Slideshow application with Test Driven Development

No more boring todo apps! Or learning non-Vue things! It's time to make a real app. We will use TDD, *test driven development,* to ensure the application functions correctly, and because we are good developers; in this day and age, with the amount of tools, frameworks and resources for testing, there is really no reason to not do it.

4.1: Let's Get Going: Setup

The vue-cli - *command line interface* - allows us to set up an entire project, with linting, tests, and everything, in a single line. Install it by opening your terminal and running `npm -g install vue-cli`. There are a handful of templates available. We will use the fully featured webpack template. Using your terminal navigate to where you want to create the project and run `vue init webpack slideshow-app`. Answer the questions as such:

```
Project name powerpoint-app
Project description Powerpoint app.
Author [Your name]
Vue build standalone
Install vue-router? No
Use ESLint to lint your code? Yes
Pick an ESLint preset Standard
Setup unit tests with Karma + Mocha? Yes
Setup e2e tests with Nightwatch? Yes
```

then `cd` in, run `yarn install` or `npm install` if you wish. Yarn will likely become the standard, so it's good to be in habit of using it, but the commands do the same thing in this case: install the modules from `package.json`. Wait a while and `npm run dev`. The application should automatically open! If not, use your browser to visit localhost:8080, where you can see the app.

4.2: Project Structure and Components Setup

This section briefly explains the structure of the project and the features we will be using. Navigate to the project folder, where you see the below structure.

```
▸ build/            // output when you build the app for production

▸ config/           // configuration for webpack, test framework, etc
▸ node_modules/     // packages
▸ src/              // the vue app goes here. The majority of the your time will be sp
ent here.
▸ static/           // static assets like images.
▸ test/             // test - contains e2e and unit - more on this soon
index.html
npm-debug.log
package.json
README.md
yarn.lock
```

Most of our time will be adding things to *src, _which is where the Vue app lives, and _test*, where the tests go. Don't worry about the others for now.

The webpack template comes with a ton of really nice features. The first **hot reloading**. When you make a change, webpack will automatically update the page, without the need to refresh. Assuming you still have the terminal open where you ran `npm run dev`, try opening `Hello.vue` which is inside of `src/components`. You should see three sections:

```
// Hello.vue
<template>
<!-- lots of html -->
<script>
export default {
  name: 'hello',
    data () {
     return {
        msg: 'Welcome to Your Vue.js App'
      }
    }
  }
}
</script>

</style scoped>
/* css */
</style>
```

This should look similar to the todo project from chapter 2 - a Vue component, all in a single `.vue` file. Try changing the `msg` variable to something different and save the file - you should see the change reflected immediately in your browser! Webpack is watching the files, and whenever you make a change, it will update the page (or display an error for you to fix).

Another important feature is the test framework included. There are two kinds of tests we will write: *unit* and *end-to-end (e2e).* Unit tests focus on testing individual pieces of code, and e2e tests look at the bigger picture, to see if lots of pieces of code are working together correctly.

Open another terminal and navigate to the root directory of the project and run `npm run unit`. This will run the unit tests - although we didn't write any yet, the template includes one to show us how they work. If you changed the `msg` variable earlier, your terminal will display a ton of errors after running `npm run unit`. Why? Scroll up a little and there should be something like the below output:

```
Hello.vue
    ✗ should render correct contents
    expected 'Welcome!' to equal 'Welcome to Your Vue.js App'
```

In my case, I changed `msg` to be 'Welcome!'; the test was expecting the original 'Welcome to Your Vue.js App' text. Let's fix it. Open `test/unit/specs/Hello.spec.js`. It looks like this:

```
import Vue from 'vue'
import Hello from '@/components/Hello'

describe('Hello.vue', () => {
  it('should render correct contents', () => {
    const Constructor = Vue.extend(Hello)
    const vm = new Constructor().$mount()
    expect(vm.$el.querySelector('.hello h1').textContent)
      .to.equal('Welcome to Your Vue.js App')
  })
})
```

The import line is `expect(vm.$el.querySelector('.hello h1').textContent).to.equal('Welcome to Your Vue.js App')`. From the start:

`vm` is the Hello component, and using the `$el.querySelector` method, we find a class called hello `.hello` containing a h1 containing the textContent we expect to see. Because it was different, the test failed. Update the test, replacing `'Welcome to Your Vue.js App'` with the message you wrote earlier in `Hello.vue`. Now run the unit test suite again using `npm run unit`.

If everything when will you should see the following output:

```
Hello.vue
    ✓ should render correct contents

PhantomJS 2.1.1 (Mac OS X 0.0.0): Executed 1 of 1 SUCCESS (0.013 secs / 0.007 secs)
TOTAL: 1 SUCCESS
```

Great! Now open `src/App.vue` and remove the line `<img src="./assets/logo.png">`.We don't need that. Also remove everything inside of `<style scoped></style>`. Your `App.vue` should now look like:

```
<template>
  <div id="#app">
    <router-view></router-view>
  </div>
</template>

<script>
export default {
  name: 'app'
}
</script>

<style scoped>
</style>
```

Time to start building. We are going to make a simple clone of Microsoft's Powerpoint application. On the left hand side of the screen, we will have a number of small, editable slides, and selecting one will show a large version of it on the right. We will write tests for each feature and piece of functionality as we go.

We will start with the left hand side bar, which will contain a number of small slides. First things first, make a component called `SlideThumbnailContainer.vue`, inside of `components`, with the following skeleton.

```
<template>
  <div class="container">
    Slides
  </div>
</template>

<script>
export default {
  name: 'SlideThumbnailContainer'
}
</script>

<style scoped>
</style>
```

You may want to keep a copy of this template, since we will use it for every component we make.

Next, make sure everything is working by importing the component in `Hello.vue`, and registering it, and rendering it, as below:

```
<template>
  <div>
    <SlideThumbnailContainer /> <!-- 3) render the component -->
  </div>
</template>

<script>
import SlideThumbnailContainer from '@/components/SlideThumbnailContainer' // 1) impor
t the components
export default {
  name: 'hello',
  components: {
    SlideThumbnailContainer // 2) register the component
  },
  data () {
    return {
      msg: 'Welcome to Your Vue.js App'
    }
  }
}
</script>

<!-- Add "scoped" attribute to limit CSS to this component only -->
<style scoped>

</style>
```

Note I have removed the default styling.

If you are still running `npm run dev` in a terminal window from before, the page should automatically update and display "Slides" on the screen (or an error overlay from the linting). Webpack hot reload is not perfect, so you might need to refresh the page. If you stopped the terminal from earlier, run `npm run dev` again.

Next, onto the thumbnails. Repeat the same procedure above: make a file called `SlideThumbnail.vue` inside `components`, put the default template in, but this time import `SlideThumbnail` inside of `SlideThumbnailContainer`, not `Hello`. And then do it again, for a component called `MainSlide`, which _will _be rendered inside of `Hello`. The current code for all the components (total of four), is below.

`MainSlide.vue`

```
<template>
  <div class="main slide container">
    Main Slide
  </div>
</template>

<script>
export default {
  name: 'MainSlide'
}
</script>

<style scoped>
.main.slide.container {
  border: 1px dotted pink;
  margin-left: 5px;
  width: 30em;
  height: 20em;
}
</style>
```

`SlideThumbnail.vue`

```
<template>
  <div class="thumbnail">
    One slide.
  </div>
</template>

<script>
export default {
  name: 'SlideThumbnail'
}
</script>

<style scoped>
.thumbnail {
  border: 1px dotted grey;
  height: 10em;
}
</style>
```

`SlideThumbnailContainer.vue`

```
<template>
  <div class="container">
    Slides
    <SlideThumbnail />
    <SlideThumbnail />
    <SlideThumbnail />
    <SlideThumbnail />
  </div>
</template>

<script>
import SlideThumbnail from '@/components/SlideThumbnail'
export default {
  name: 'SlideThumbnailContainer',
  components: {
    SlideThumbnail
  }

}
</script>

<style scoped>
.container {
  display: flex;
  justify-content: space-around;
  flex-direction: column;
  width: 15em;
  height: auto;
}
</style>
```

`Hello.vue`

```
<template>
  <div class="hello">
    <SlideThumbnailContainer />
    <MainSlide />
  </div>
</template>

<script>
import SlideThumbnailContainer from '@/components/SlideThumbnailContainer'
import MainSlide from '@/components/MainSlide'
export default {
  name: 'hello',
  components: {
    SlideThumbnailContainer,
    MainSlide
  },
  data () {
    return {
      msg: 'Welcome to Your Vue.js App'
    }
  }
}
</script>

<style scoped>
.hello {
  display: flex;
  justify-content: stretch;
}
</style>
```

I added some css so the application looks decent -- css is not the focus of this book, so it is not explained. It's fairly simple css anyway, so it should be easy enough to figure out things are working. Notice I use the same class, `container`, multiple times - because of the `scoped` tag in the `<style>` tag, the style is uniquely applied to the component it is declared in.

Your browser should now show...

[TODO: Image]

Not too useful. However, using a few directives, we can do a lot. But first, we need to have a brief talk about state management in Vue, and the _flux _pattern that was born to handle complex user interfaces in single page applications.

4.3: Unidirectional flow, and the single source of truth

Managing data for a complex single page application can get very difficult, very fast. The solution eventually arrived on to handle this is the *flux _pattern, popularized by Facebook. In flux, your application has a _store* - literally, a huge Javascript object that stores all the data your application needs. All the components go to the store for data, so the store is known as the "single source of truth" - the only place data is stored. There is a little more to flux that will be introduced later using Vue's very own *vuex* module, a flux implementation with deep integration with Vue, but for now the important thing to know is we should store all the data in a single location, and pass bits and pieces to components. Since Vue is reactive, whenever something changes in the store (in the pointpoint app, an object contained in a top level `data` function) the changes will be automatically reflected in all the other components.

4.4: Creating the Store

We will create our store object in `Hello`, since it is the top level component in our application. Update `Hello.vue` as shown below:

Hello.vue

```
<template>
  <div class="hello">
    <SlideThumbnailContainer />
    <MainSlide />
  </div>
</template>

<script>
import SlideThumbnailContainer from '@/components/SlideThumbnailContainer'
import MainSlide from '@/components/MainSlide'
export default {
  name: 'hello',
  components: {
    SlideThumbnailContainer,
    MainSlide
  },
  data () {
    return {
      msg: 'Welcome to Your Vue.js App',
      store: {
        slides: [
          { id: 0, title: 'Demo', content: 'This is a demo slide' },
          { id: 1, title: 'Vue', content: 'Flux is a great way to manage your app' }
        ]
      }
    }
  }
}
</script>

<style scoped>
.hello {
  display: flex;
  justify-content: stretch;
}
</style>
```

Looks good. Now, let's pass the slides to the `SlideThumbnailContainer` to display - but before doing so, let's write our first real test, to ensure the behaviour is as expected.

4.5: Test! SlideThumbnailContainer.spec.js

Inside test/unit/specs/, create a new file called `SlideThumbnailContainer.spec.js`. Next, add the following test skeleton. Again, it might be a good idea to keep a copy of this - all tests start out using the same template.

SlideThumbnailContainer.spec.js

```
import Vue from 'vue'
import SlideThumbnailContainer from '@/components/SlideThumbnailContainer'

describe('SlideThumbnailContainer.vue', () => {
  it('should receieve an array of slides as a prop', () => {

  })
})
```

First we import Vue, and the components, and whatever else we need to test. In this case, we will be mounting the component, and using some of the Vue internal apis, so we need both Vue and the component. If we were just testing a method that does not need Vue itself, we would simple import the method, and nothing else.

The `describe` block explains what the subject of this spec file is, and the `it` block (which we can have any amount of) explains what each individual test is doing.

First things first - we should declare and extend a Vue instance with our component, SlideThumbnailContainer. This way, the component will get all the Vue lifecycle and api methods, such as `created`, `data`, `props`, etc, as it does in the real application, where we import the component and attach it in the `components: { }` section of `Hello.vue`.

```
const Component = Vue.extend(SlideThumbnailContainer)
```

Next, we instantiate an instance of the component. At this point, we can pass props using the `propsData` option. You can read more about this api option here: https://vuejs.org/v2/api/#propsData

```
const vm = new Component({
  propsData: {
    slides: [
      { id: 0, content: 'Test' }
    ]
  }
})
```

We want to test that a `slides` prop is passed. Lastly, we mount the component using the lifecycle hook `$mount`. This is called automatically in the main application, when Vue detects custom component markup, such as `<SlideThumbnailContainer />`, however in a unit test to trigger rendering, so we need to do it manually.

```
vm.$mount()
```

Now, time to make some assertions about what we want the component to look like after it has been mounted. We expect it to have a `slides` prop, which has one item in it, with `content` equal to `'Test'`.

```
expect(vm.slides.length).to.equal(1)
expect(vm.slides[0].content).to.equal('Test')
```

The full listing looks like:

SlideThumbnailContainer.spec.js

```
import Vue from 'vue'
import SlideThumbnailContainer from '@/components/SlideThumbnailContainer'

describe('SlideThumbnailContainer.vue', () => {
  it('should receieve an array of slides', () => {
    const Component = Vue.extend(SlideThumbnailContainer)
    const vm = new Component({
      propsData: {
        slides: [
          { id: 0, content: 'Test' }
        ]
      }
    })
    vm.$mount()

    expect(vm.slides.length).to.equal(1)
    expect(vm.slides[0].content).to.equal('Test')
  })
})
```

Now, run the test. We expect it to fail - we haven't implemented the functionality yet! The errors *should* help guide us.

`npm run unit`

I get:

```
✘ 1 problem (1 error, 0 warnings)

Errors:
  1  http://eslint.org/docs/rules/comma-dangle
 @ ./test/unit/specs/SlideThumbnailContainer.spec.js 7:31-78
 @ ./test/unit/specs \.spec$
 @ ./test/unit/index.js

  SlideThumbnailContainer.vue
    ✗ should receieve an array of slides
    undefined is not an object (evaluating 'vm.slides.length')
    webpack:///test/unit/specs/SlideThumbnailContainer.spec.js:16:21 <- index.js:10570
:21


PhantomJS 2.1.1 (Mac OS X 0.0.0): Executed 1 of 1 (1 FAILED) ERROR (0.035 secs / 0.01
secs)
```

`vm.slides.length` is `undefined`. Why? We didn't declare any props in the component! Let's fix that.

ThumbnailContainer.vue

```
<template>
  <div class="container">
    Slides
    <SlideThumbnail />
    <SlideThumbnail />
    <SlideThumbnail />
    <SlideThumbnail />
  </div>
</template>

<script>
import SlideThumbnail from '@/components/SlideThumbnail'
export default {
  name: 'SlideThumbnailContainer',
  components: {
    SlideThumbnail
  },
  props: ['slides']
}
</script>

<style scoped>
/* omitted */
</style>
```

Now `ThumbnailContainer.vue` knows to receive a `slides` prop. Another way to write props is:

```
props: {
  slides: {
    type: Object,
    default () {
      return [{ id: 0, content: 'default content' }]
    }
  }
}
```

This way is a bit more typing, and takes a little more space, but is sometimes preferred, especially in large projects, where other developers might not know what kind of object will be passed. I prefer the second way, since it serves as a kind of documentation for your component - all the information, such as what the component does and what kind of data it needs to do it's job is self contained. For now I will use the first one, for sake of saving space.

Run `npm run unit` again...

```
SlideThumbnailContainer.vue
    ✓ should receieve an array of slides

PhantomJS 2.1.1 (Mac OS X 0.0.0): Executed 1 of 1 SUCCESS (0.019 secs / 0.01 secs)
TOTAL: 1 SUCCESS
```

Great! The test passes - we can be confident the component is receiving the right data.

> Note: technically for this particular test, `$mount` is not needed, since we are not rendering anything. Props can be passed without any need to render or even enter the Vue lifecycle - however, the next thing will do is render something (the slide thumbanils) so it felt like a good time to introduce it.

Okay - we have the data. Now, let's render it. At the moment, if you insert a `console.log(vm.$el)` into the test after `vm.$mount()`, you can see the rendered markup. Namely:

```
<div data-v-5198ddbe="" class="container">
  Slides
  <div data-v-3824ac9a="" data-v-5198ddbe="" class="container">
  One slide.
</div> <div data-v-3824ac9a="" data-v-5198ddbe="" class="container">
  One slide.
</div> <div data-v-3824ac9a="" data-v-5198ddbe="" class="container">
  One slide.
</div> <div data-v-3824ac9a="" data-v-5198ddbe="" class="container">
  One slide.
</div></div>
```

Pretty ugly - the HTML formatting isn't that nice, but you can see are just rendering a bunch of hard coded `SlideThumbnail` components. What we want to do is loop through the `slides` passed as a prop, and render those - like we did in the Todo application.

We can achieve this using like so:

```
<template>
  <div class="container">
    Slides
    <SlideThumbnail v-for="slide in slides" :slide="slide" key="slide.id" />
  </div>
</template>

<script>
  ...
</script>

<style scoped>
  ...
</style>
```

Using `v-for` and passing each element to the `SlideThumbnail` component. Note, the `...` means the content has not changed, in `<script>` and `<style>`.

`npm unit test` yields:

```
<div data-v-5198ddbe="" class="container">
  Slides
  <div data-v-3824ac9a="" data-v-5198ddbe="" class="container" slide="[object Object]"
>
  One slide.
</div></div>

  SlideThumbnailContainer.vue
    ✓ should receieve an array of slides
```

`v-for` did it's job, and the `slide` object is indeed been passed to the `SlideThumbnail` component - however nothing is happening. This make sense - we did not implement any logic for `SlideThumbnail` yet. The test still passes, since we did not make any expectations about what to render. Let's do that first.

After the two current `expect` asserts, add another:

```
expect(vm.$el.querySelector('.thumbnail').textContent.trim()).to.equal('Test')
```

We can select an element by class using `querySelector('.thumbnail')`, since the class name is `thumbnail`. If it is an id, we do `'#thumbnail'`. `trim()` is required to remove any whitespace, which some browsers add. Running the test using `npm run unit` yields:

```
LOG LOG: <div data-v-5198ddbe="" class="container">
  Slides
  <div data-v-3824ac9a="" data-v-5198ddbe="" class="thumbnail" slide="[object Object]"
>
  One slide.
</div></div>

  SlideThumbnailContainer.vue
    ✗ should receieve an array of slides
    expected 'One slide.' to equal 'Test'
```

To get this test to pass, we need to implement the functionality for `SlideThumbnail`. This includes receiving the correct prop (from `ThumbnailContainer`, and then rendering the content).

4.6: Another Test! SlideThumbnail.spec.js

First a test for the prop. Create `test/unit/spec/SlideThumbnail.spec.js`, and write a test for the prop. It should be almost the same as the previous test, but with the correct component. I'd recommend trying to write it without immediately looking below at the code. There is a small gotcha! So don't spent too long if it isn't working.

Here is how the test looks:

test/unit/spec/SlideThumbnail.spec.js

```
import Vue from 'vue'
import SlideThumbnail from '@/components/SlideThumbnail'

describe('SlideThumbnail.vue', () => {
  it('should receieve a single slide with content', () => {
    const Component = Vue.extend(SlideThumbnail)
    const vm = new Component({
      propsData: {
        slide: { id: 0, content: 'Test' }
      }
    })
    vm.$mount()

    expect(vm.slide.content).to.equal('Test')
    expect(vm.$el.textContent.trim()).to.equal('Test')
  })
})
```

Of course it will fail at first - we need to pass the prop and render it. The updated `SlideThumbnail` looks like this:

```
<template>
  <div class="thumbnail">
    {{ slide.content }}
  </div>
</template>

<script>
export default {
  props: ['slide'],
  name: 'SlideThumbnail'
}
</script>

<style scoped>
  ...
</style
```

Now running `npm run unit` should yield *two* passing tests - by making `SlideThumbnail` pass, `SlideThumbnailContainer` is also able to get the correct content rendered and pass.

Two points of interest before we go on:

1) If you remove `vm.$mount()`, you will find the tests fail - why? Because by not mounting the component, no rendering occurs, so no content is able to be found using `querySelector` and the tests fail.

2) In the `SlideThumbnail` test, `vm.$el.querySelector` returns null. The reason is because in that component, the text we are looking for is in the *root* `<div>` tag. `querySelector` looks for elements within the root `<div>` tag, not include itself. So we can just do `vm.$el.textContent` to get the `slide.content`.

Writing these tests does take a while, but in the long run it is worth it. You can know if a new feature breaks old ones, or if another developer works on your application in the future, it will be much easier for them to know how the app works, and not break existing features. It is a good habit to get into and second nature to good software developers.

4.7: Checkpoint

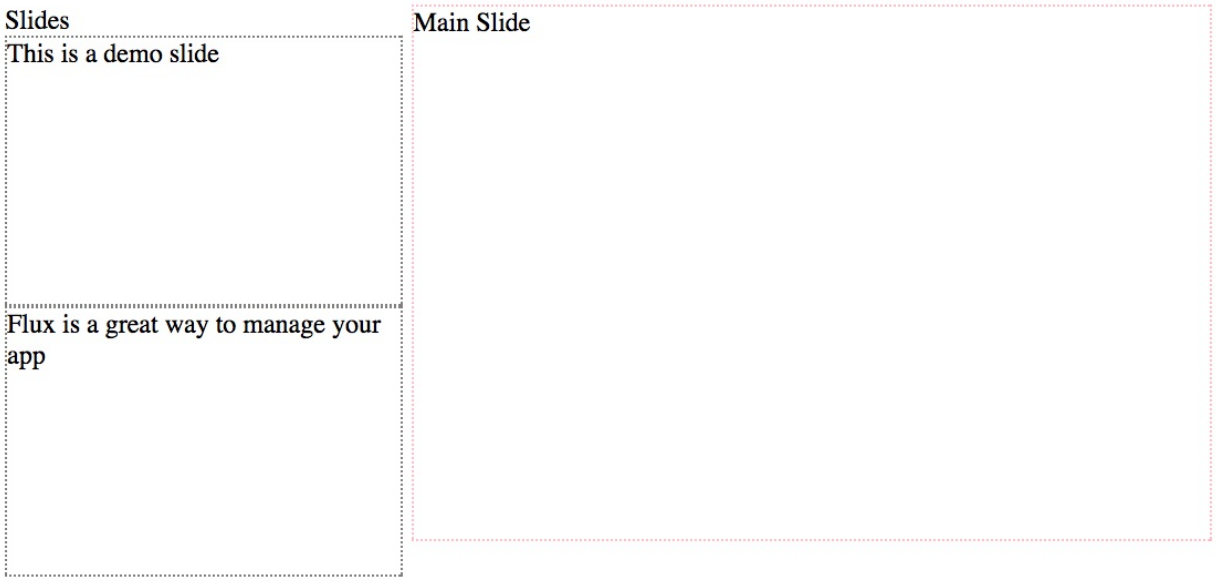
Let's actually see what the app looks like in the browser. Don't forget to actually pass the `store.slides` to the `SlideThumbnailContainer` in `Hello`, like below:

```
<template>
  <div class="hello">
    <SlideThumbnailContainer :slides="store.slides" />
    <MainSlide />
  </div>
</template>

<script>
import SlideThumbnailContainer from '@/components/SlideThumbnailContainer'
import MainSlide from '@/components/MainSlide'
export default {
  name: 'hello',
  components: {
    SlideThumbnailContainer,
    MainSlide
  },
  data () {
    return {
      msg: 'Welcome to Your Vue.js App',
      store: {
        slides: [
          { id: 0, title: 'Demo', content: 'This is a demo slide' },
          { id: 1, title: 'Vue', content: 'Flux is a great way to manage your app' }
        ]
      }
    }
  }
}
</script>

<style scoped>
.hello {
  display: flex;
  justify-content: stretch;
}
</style>
```

I see something like below:



4.8: The first integration test

The next feature will be when clicking on a thumbnail, you the main slide should show the content, but enlarged, and allow you to edit it. Since we are good software developers, let's write a test for that! Firstly another unit test, to make sure the `MainSlide` is getting the write values passed, then another kind of test - an integration, or end to end (or *e2e* for short) test.

The unit test starts of looks similar to the previous ones:

MainSlide.spec.js

```
import Vue from 'vue'
import MainSlide from '@/components/MainSlide'

describe('MainSlide.vue', () => {
  it('should receieve a single slide with content', () => {
    const Component = Vue.extend(MainSlide)
    const vm = new Component({
      propsData: {
        slide: { id: 0, content: 'Test' }
      }
    })
    vm.$mount()

    expect(vm.$el.querySelector("#content").value).to.equal('Test')
  })
})
```

`MainSlide` does not just show content - it lets us *edit* it, too. We will use `<input>` tags, and `v-model`. This test fails - try to update `MainSlide` yourself, before looking below to see how to get the test to pass.

MainSlide.vue

```
<template>
  <div class="main slide container">
    <input id="content" type="text" v-model="slide.content">
  </div>
</template>

<script>
export default {
  name: 'MainSlide',
  props: ['slide']
}
</script>

<style scoped>
  ...
</style>
```

The test should pass now! However, if you try the app out in your browser, the console will throw errors such as:

```
[Vue warn]: Error in render function: "TypeError: Cannot read property 'content' of un
defined"

found in

---> <MainSlide> at ...
```

The test is fine -- because we pass a slide using `propsData`. However in the actual app, we do not pass a slide, so `slide.content` is undefined, and `v-model` doesn't know what to bind too. The goal is to click a `SlideThumbnail`, and then have that reflected in `MainSlide`. Let's use TDD to assist in creating this feature. This test will come in two parts - the unit tests, for individual methods in `SlideThumbnail`, `SlideThumbnailContainer` and `Hello`, and an integration test, to actually simulate clicking a slide in a browser.

A quick review of the architecture of our app:

- Hello
  - MainSlide
  - SlideThumbnailContainer
    - SlideThumbnail

Remembering all the data is stored at the `Hello` level. When `SlideThumbnail` is clicked, we want to set a new variable called `mainSlide` at the top level, based on the thumbnail clicked, which will the be passed to `MainSlide`. To achieve this we will use the `$emit` API method, like in the Todo app in chapter 2. In this case, the `id` of the selected slide will be emitted from `SlideThumbnail` to `SlideThumbnailContainer`, and then again from `SlideThumbnailContainer` to `Hello`. First, a unit test on `SlideThumbnail` for the existence of such a method. Inside of SlideThumbnail.spec.js, under the first `it()` statement, add:

```
it('should have a function to emit an event when clicked', () => {
  const vm = new Vue(SlideThumbnail)

  expect(vm.$options.methods.clicked).to.be.a('function')
})
```

This test simply verifies the existence of the required method, which I am calling `clicked`. Since the feature involves many components, an integration test is more appropriate to ensure the correct behavior, however is it still good to verify the function exists - this kind of test could help a future developer when it comes to refactoring, or understanding what a function is used for.

To get this to pass, update the component to include the method as below:

SlideThumbnail.vue

```
<template>
  <div class="thumbnail">
    {{ slide.content }}
  </div>
</template>

<script>
export default {
  props: ['slide'],
  name: 'SlideThumbnail',
  methods: {
    clicked () {
      this.$emit('slideSelected', this.slide.id)
    }
  }
}
</script>

<style scoped>
...
</style>
```

A similar test is needed in SlideThumbnailContainer.spec.js

```
it('should contain a function to emit a clicked slide', () => {
  const vm = new Vue(SlideThumbnailContainer)

  expect(vm.$options.methods.slideEmitted).to.be.a('function')
})
```

And to get it to pass, add a `methods` object with the required method:

SlideThumbnailContainer.vue

```
methods: {
  slideEmitted (id) {
    this.$emit('setMainSlide', id)
  }
}
```

Now lastly, a method in the top level `Hello` component is required, to actually set the main slide, as well as a variable in the store object to hold it. I deleted the `Hello.spec.js` the template initially created, but I'll recreate it now, and add a test for a method that listens for the `setMainSlide` event, and finds the correct slide by the `id` passed from the event.

Hello.spec.js

```
import Vue from 'vue'
import Hello from '@/components/Hello'

describe('Hello.vue', () => {
  it('should listen for a setMainSlide event and set the correct slide', () => {
    const Component = Vue.extend(Hello)
    const vm = new Component({
      propsData: {
        mainSlide: null
      }
    }).$mount()

    vm.store = {
      slides: [
        { id: 0, content: 'Text' }
      ]
    }

    expect(vm.setMainSlide).to.be.a('function')

    vm.setMainSlide(0)

    expect(vm.mainSlide).to.not.equal(null)
    expect(vm.mainSlide.content).to.equal('Text')
  })
})
```

Notice that *after* calling `$mount` I set `vm.store` to have a single slide. If you pass the `store` when making the extending the initial Vue instance, instead of using the one you pass, the one defined in the original component will take priority. In this case, instead of getting the single slide we would be passing in this test, the component would use the two hardcoded slides in the store we defined in the `.vue` file earlier. Try moving the store inside the `new Component` declaration and see what happens.

Getting this test to pass it also trivial; just add the `mainSlide` prop and a simple method that filters and assigns `mainSlide` for the correct slide based on `id`.

```
<template>
  <div class="hello">
    <SlideThumbnailContainer :slides="store.slides" />
    <MainSlide />
  </div>
</template>

<script>
import SlideThumbnailContainer from '@/components/SlideThumbnailContainer'
import MainSlide from '@/components/MainSlide'
export default {
  name: 'hello',
  components: {
    SlideThumbnailContainer,
    MainSlide
  },
  data () {
    return {
      store: {
        slides: [
          { id: 0, title: 'Demo', content: 'This is a demo slide' },
          { id: 1, title: 'Vue', content: 'Flux is a great way to manage your app' }
        ]
      },
      mainSlide: null
    }
  },
  methods: {
    setMainSlide (id) {
      this.mainSlide = this.store.slides.filter(s => s.id === id)[0]
    }
  }
}
</script>

<style scoped>
...
</style>
```

`npm run unit` shows all our tests passing. Each component works individually. Even though we have methods for emitting events, we haven't wired the event listeners up, so the components cannot communicate.. *yet.* This is what integration tests are for - to ensure many components can combine their functionality and produce a complete feature.

Start by deleting the integration test the template came with in `test/e2e/specs/Hello.js`. Next, create a file called `SetMainSlide.js`. This is where our test will go. Inside, add the following:

```
module.exports = {
  'Clicking a thumbnail sets the main slide': function (browser) {
    const devServer = browser.globals.devServerURL

    browser
      .url(devServer)
      .waitForElementVisible('#app', 5000)
      .assert.elementNotPresent('.main.slide.container')
      .click('.thumbnail')
      .assert.elementPresent('.main.slide.container')
      .getValue('.main.slide.container #content', function (result) {
        this.assert.equal(result.value, 'This is a demo slide')
      })
  }
}
```

Integration tests should generally read as if a user is actually interacting with the app. In this test, we wait for the app to render, and asserts that the main slide is *not* present - since no thumbnail has been clicked yet. After clicking on a thumbnail, we asserts that it *should* be present, and contain the value of the first slide.

Running this test using `npm run e2e` yields something like this:

```
TEST FAILURE:  1 assertions failed, 3 passed. (4.122s)

 ✖ SetMainSlide

   - Clicking a thumbnail sets the main slide (2.426s)
   Testing if element <.main.slide.container> is present.  - expected "present" but go
t: "not present"
```

Of course, because we have not set up the event listeners. Let's do so, start from `SlideThumbnail.vue`.

```
<template>
  <div class="thumbnail" @click="clicked"> <!-- Listen for a click -->
    {{ slide.content }}
  </div>
</template>

<script>
export default {
  props: ['slide'],
  name: 'SlideThumbnail',
  methods: {
    clicked () {
      this.$emit('slideSelected', this.slide.id)
    }
  }
}
</script>

<style scoped>
...
</style>
```

Next, `SlideThumbnailContainer` needs to listen for `slideSelected`, and in turn emit `setMainSlide`.

```
<template>
  <div class="container">
    Slides
    <SlideThumbnail
       v-for="slide in slides"
      :slide="slide"
      @slideSelected="slideEmitted"
      key="slide.id" />
  </div>
</template>

<script>
import SlideThumbnail from '@/components/SlideThumbnail'
export default {
  name: 'SlideThumbnailContainer',
  components: {
    SlideThumbnail
  },
  props: ['slides'],
  methods: {
    slideEmitted (id) {
      this.$emit('setMainSlide', id)
    }
  }
}
</script>

<style scoped>
...
</style>
```

Lastly, `Hello` will listen for `setMaiSlide`, and allocate the correct slide based on the id. I also will pass the `mainSlide` prop to `MainSlide`, and conditionally render `MainSlide` using `v-if` - we don't want to display it if no slide is selected.

```
<template>
  <div class="hello">
    <SlideThumbnailContainer
      :slides="store.slides"
      @setMainSlide="setMainSlide"
    />
    <MainSlide :slide="mainSlide" v-if="mainSlide" />
  </div>
</template>

<script>
import SlideThumbnailContainer from '@/components/SlideThumbnailContainer'
import MainSlide from '@/components/MainSlide'
export default {
  name: 'hello',
  components: {
    SlideThumbnailContainer,
    MainSlide
  },
  data () {
    return {
      store: {
        slides: [
          { id: 0, title: 'Demo', content: 'This is a demo slide' },
          { id: 1, title: 'Vue', content: 'Flux is a great way to manage your app' }
        ]
      },
      mainSlide: null
    }
  },
  methods: {
    setMainSlide (id) {
      this.mainSlide = this.store.slides.filter(s => s.id === id)[0]
    }
  }
}
</script>

<style scoped>
...
</style>
```

4.9: An add slide button

To finish the app, we need a way to add and remove slides. The tests and techniques are nothing not already covered, but serve as a good way to practise TDD with Vue. First, a test for the existence of a addSlide method:

SlideThumbnailContainer.spec.js

```
import Vue from 'vue'
import SlideThumbnailContainer from '@/components/SlideThumbnailContainer'

describe('SlideThumbnailContainer.vue', () => {

  ...

  it('should contain a function to create a new slide when clicked', () => {
    const vm = new Vue(SlideThumbnailContainer)

    expect(vm.$options.methods.addSlide).to.be.a('function')
  })
})
```

Satisfying the test (and adding a button to trigger the method).

SlideThumbnailContainer.vue

```
<template>
  <div class="container">
    Slides
    <button
      id="add-slide-button"
      @click="addSlide">
        Add Slide
    </button>
    <SlideThumbnail
       v-for="slide in slides"
      :slide="slide"
      @slideSelected="slideEmitted"
      key="slide.id" />
  </div>
</template>

<script>
import SlideThumbnail from '@/components/SlideThumbnail'
export default {
  name: 'SlideThumbnailContainer',
  components: {
    SlideThumbnail
  },
  props: ['slides'],
  methods: {
    slideEmitted (id) {
      this.$emit('setMainSlide', id)
    },
    addSlide () {
      this.$emit('addSlide')
    }
  }
}
</script>

<style scoped>
...
</style>
```

Next, `Hello` needs to listen for this event, and add a new slide. The test is as follows:

```
import Vue from 'vue'
import Hello from '@/components/Hello'

describe('Hello.vue', () => {
  it('should listen for a setMainSlide event and set the correct slide', () => {
    //...
  })

  it('should add a new slide', () => {
    const Component = Vue.extend(Hello)
    const vm = new Component({
      propsData: {
        mainSlide: null
      }
    })

    vm.store = {
      slides: []
    }

    expect(vm.addSlide).to.be.a('function')

    vm.addSlide()
    vm.addSlide()

    expect(vm.store.slides.length).to.equal(2)
    expect(vm.store.slides[0].id).to.equal(1)
    expect(vm.store.slides[1].id).to.equal(2)
  })
})
```

Fairly straight forward. Add two slides, asserts that the slide array has a length of 2, and `slide.id` is 0 and 1 respectively.

The implementation is as follows (a good exercise is to get this test to pass before looking, though).

```
<template>
  <div class="hello">
    <SlideThumbnailContainer
      :slides="store.slides"
      @setMainSlide="setMainSlide"
      @addSlide="addSlide"
    />
    <MainSlide :slide="mainSlide" v-if="mainSlide" />
    {{ store.slides }}
  </div>
</template>

<script>
```

```
import SlideThumbnailContainer from '@/components/SlideThumbnailContainer'
import MainSlide from '@/components/MainSlide'
export default {
  name: 'hello',
  components: {
    SlideThumbnailContainer,
    MainSlide
  },
  data () {
    return {
      store: {
        slides: [
          { id: 0, title: 'Demo', content: 'This is a demo slide' },
          { id: 1, title: 'Vue', content: 'Flux is a great way to manage your app' }
        ]
      },
      mainSlide: null
    }
  },
  methods: {
    setMainSlide (id) {
      this.mainSlide = this.store.slides.filter(s => s.id === id)[0]
    },
    addSlide () {
      let _id = this.store.slides.length === 0 ? 1 : this.getNextId()
      this.store.slides.push({
        id: _id,
        content: ''
      })
    },
    getNextId () {
      return Math.max(...this.store.slides.map(a => a.id)) + 1
    }
  }
}
</script>

<style scoped>
...
</style>
```

There is a little bit of trickiness going on to get the id for the slide. If there is nothing in `store.slides`, we start at 1. Else, find the largest `id` and add one to it.

All the tests should be passing after running `npm run unit`. To finish this feature off, an integration test is in order. This one will test the full workflow - starting with a blank slideshow, the user should be able to click the "Add Slide" button, then click on the thumbnail, and update the text.

```
module.exports = {
  'Clicking the add button adds a new slide': function (browser) {
    const devServer = browser.globals.devServerURL

    browser
      .url(devServer)
      .waitForElementVisible('#app', 5000)
      .assert.elementNotPresent('.main.slide.container')
      .click('#add-slide-button')
      .click('.thumbnail')
      .assert.elementPresent('.main.slide.container')
      .setValue('#content', 'Some content')
      .getText('.thumbnail', function (result) {
        this.assert.equal(result.value, 'Some content')
       })
  }
}
```

Running `npm run e2e` yields a bunch of errors at this point. The output can be a bit verbose, but you should see something like this:

```
Running:  Clicking the add button adds a new slide
 ✔ Element <#app> was visible after 88 milliseconds.
 ✔ Testing if element <.main.slide.container> is not present.
 ✔ Testing if element <.main.slide.container> is present.
 ✖ Failed [equal]: ('This is a demo slideSome content' == 'Some content')  - expected
"Some content" but got: "This is a demo slideSome content"
    at Object.<anonymous> (/Users/lachlan/vuejs-book/chapter4/powerpoint/test/e2e/spec
s/AddANewSlide.js:14:21)
```

What is happening? A new slide is created, and when we do `click(.thumbnail)` it is actually selecting the first `<div>` with the thumbnail class, which is actually one of the hardcoded slides we set up earlier in `Hello`, since we had not functionality to add a slide. Now we do, however - so update `Hello.vue` to have the following store:

```
data () {
  return {
    store: {
      slides: []
    },
    mainSlide: null
  }
}
```

The AddASlide test should now pass, but the original test we wrote, SetMainSlide, now fails - because there is no slide to set, since we removed the hardcoded ones from the store. All the functionality SetMainSlide was testing - the ability to click a thumbnail, then have `MadeSlide.vue` render it is now included in AddASlide, so we can delete SetMainSlide - it is no longer needed. Sometimes, refactoring or even deleting unnecessary code is as important as developing new features.

Deleting SetMainSlide and running `npm run test` should execute all the unit and e2e tests, of which all should be passing.

There you have it - a simple slideshow app, developing using unit and integration tests, which is maintainable and easily extendable. An additional feature which will be left as an exercise will be the ability to delete a slide.