Web Application Development

Produced by

David Drohan (ddrohan@wit.ie)

Department of Computing & Mathematics
Waterford Institute of Technology

http://www.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Vue.js

PART 2

COMPONENTS

Vue.js

Overall Section Outline

1. **Introduction** – Why you should be using VueJS
2. **Terminology & Overview** – The critical foundation for understanding
3. **Declarative Rendering & Reactivity** – Keeping track of changes (Data Binding)
4. **Components** – Reusable functionality (Templates, Props & Slots)
5. **Routing** – Navigating the view (Router)
6. **Directives**– Extending HTML
7. **Event Handling** – Dealing with User Interaction
8. **Filters** – Changing the way we see things
9. **Computed Properties & Watchers** – Reacting to Data Change
10. **Transitioning Effects** – I like your <style>
11. **Case Study** – Labs in action

Vue.js

Overall Section Outline

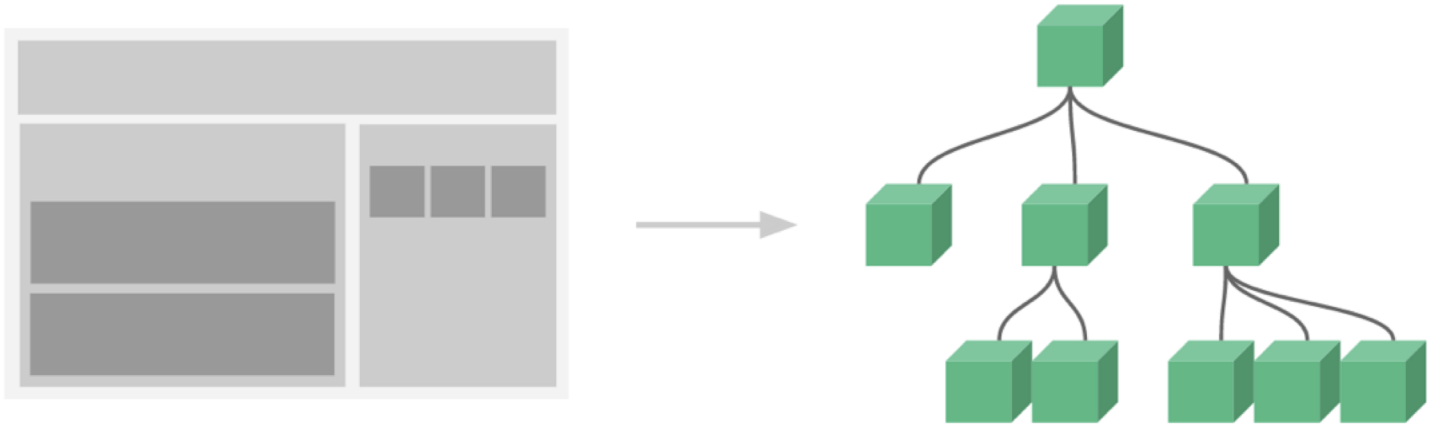
1. **Introduction** – Why you should be using VueJS
2. **Terminology & Overview** – The critical foundation for understanding
3. **Declarative Rendering & Reactivity** – Keeping track of changes (Data Binding)
4. **Components – Reusable functionality (Templates, Props & Slots)**
5. **Routing** – Navigating the view (Router)
6. **Directives**– Extending HTML
7. **Event Handling** – Dealing with User Interaction
8. **Filters** – Changing the way we see things
9. **Computed Properties & Watchers** – Reacting to Data Change
10. **Transitioning Effects** – I like your <style>
11. **Case Study – Labs in action**

Components

REUSABLE FUNCTIONALITY

Introduction - Recap

Once again, as previously mentioned, the component system is another important concept in Vue (possibly ***the*** most important concept), because it's an abstraction that allows us to build large-scale applications composed of small, self-contained, and often reusable components. If we think about it, almost any type of application interface can be abstracted into a tree of components:



Introduction - Recap

Components can be included in a single file:

```
// Define a new component called todo-item
Vue.component('todo-item', {
  template: '<li>This is a todo</li>'
})
```

```
<ol>
  <!-- Create an instance of the todo-item component -->
  <todo-item></todo-item>
</ol>
```

Introduction - Recap

Or modularized into their own **.vue** files
(which is what we'll do)

Vue.js

Components in Depth

In Vue, components are reusable Vue instances and need to have at least 2 things:

- A name (obvious)
- A template (The rendered DOM that belongs to each component)

and *because* they are reusable Vue instances they also accept the same options i.e.

- **`data`**
- **`computed`**
- **`watch`**
- **`methods`** and
- **lifecycle hooks** (**`mounted`**, **`unmounted`**) etc.

Vue.js

Reusing Components

Here's an example of a Vue Component (in-line) :

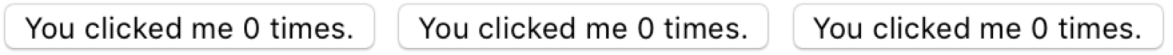
```
Vue.component('button-counter', {
        data: function () {
                return {
                  count: 0
                }
        },
        template: '<button v-on:click="count++">You clicked me {{ count }}
                        times.</button>'
})
```

Reusing Components

Vue Components can be reused as often as you like:

```
<div id="components-demo">
  <button-counter></button-counter>
  <button-counter></button-counter>
  <button-counter></button-counter>
</div>
```

Result in Browser



If you were to click on the buttons, each one will maintain its own, separate **count**. That's because each time you use a component, a new **instance** of it is created.

Reusing Components & **data**

** **data** Must Be a Function **

When we defined the **<button-counter>** component, you may have noticed that **data** wasn't directly provided an object, like:

```
data: {
  count: 0
}
```

Instead, **a component's data option must be a function**, so that each instance can maintain an independent copy of the returned data object:

```
data: function () {
  return {
    count: 0
  }
}
```

or

```
data() {
    return {
          count: 0
     }
}
```

If Vue didn't have this rule, clicking on one button would affect the **data** of ***all other instances***,

Single-File Components & Registration

In-line components can work very well for small to medium-sized projects, where JavaScript is only used to enhance certain views. In more complex projects however, or when your frontend is entirely driven by JavaScript, these disadvantages become apparent:

- **Global definitions** force unique names for every component
- **String templates** lack syntax highlighting and require ugly slashes for multiline HTML
- **No CSS support** means that while HTML and JavaScript are modularized into components, CSS is conspicuously left out
- **No build step** restricts us to HTML and ES5 JavaScript, rather than preprocessors like Pug (formerly Jade) and Babel

All of these are solved by **single-file components** with a **.vue** extension, made possible with build tools such as **Webpack** or Browserify.

Single-File Components & Registration

Here's our **`AboutUs.vue`**, Now we get:

- **Complete syntax highlighting**
- **CommonJS modules**
- **Component-scoped CSS**

All of this is possible thanks to the use of **Webpack**. The Vue CLI makes this very easy and is supported out of the box. **.vue** files **cannot** be used without a webpack setup, and as such, they are not very suited to apps that just use Vue on a page without being a full-blown single-page app (SPA).

.../src/components/AboutUs.vue [donationvue]

AboutUs.vue

```vue
<template>
  <div class="hero">
    <h3 class="vue-title"><i class="fa fa-info" style="padding: 3px"></i>{{messagetitle}}</h3>
    <dfooter></dfooter>
  </div>
</template>

<script>
import Footer from '@/components/Footer'

export default {
  name: 'AboutUs',
  data () {
    return {
      messagetitle: ' About Us '
    }
  },
  components: {
    'dfooter': Footer
  }
}
</script>

<style scoped>
  .vue-title {
    margin-top: 30px;
    text-align: center;
    font-size: 45pt;
    margin-bottom: 10px;
  }
</style>
```

3:49 LF UTF-8 Git: master

Vue.js

Passing Data to Child Components (with **Props**)

We've looked at using (and reusing) components in a Vue app and the potential benefits that offers to the developer but the problem is, components won't really be useful unless you can pass data to it, such as a 'title' and 'content' for example, of a specific object, (say a 'post') we want to display. That's where **props** come in.

Props are custom attributes you can register on a component. When a value is passed to a prop attribute, it becomes a property on that component instance. To pass a 'title' to a 'blog post' component, we can include it in the list of props this component accepts, using a **props** option:

```
Vue.component('blog-post', {
  props: ['title'],
  template: '<h3>{{ title }}</h3>'
})
```

Passing Data to Child Components (with Props)

A component can have as many **props** as you'd like and by default, any value can be passed to any prop. In the template previous, you'll see that we can access this value on the component instance, just like with **data**.

Once a prop is registered, you can pass data to it as a custom attribute, like this:

```
<blog-post title="My journey with Vue"></blog-post>
<blog-post title="Blogging with Vue"></blog-post>
<blog-post title="Why Vue is so fun"></blog-post>
```

Passing Data to Child Components (with Props)

- Prop Casing (camelCase vs kebab-case)

  HTML attribute names are case-insensitive, so browsers will interpret any uppercase characters as lowercase. That means when you're using in-DOM templates, camelCased prop names need to use their kebab-cased (hyphen-delimited) equivalents:

```
Vue.component('blog-post', {
  // camelCase in JavaScript
  props: ['postTitle'],
  template: '<h3>{{ postTitle }}</h3>'
})
```

```
<!-- kebab-case in HTML -->
<blog-post post-title="hello!"></blog-post>
```

  Again, if you're using string templates, this limitation does not apply.

Passing Data to Child Components (with Props)

- Prop Types

  So far, we've only seen props listed as an array of strings, for example:

```
props: ['title', 'likes', 'isPublished', 'commentIds', 'author']
```

  Usually though, you'll want every prop to be a specific type of value. In these cases, you can list props as an object, where the properties' names and values contain the prop names and types, respectively:

```
props: {
  title: String,
  likes: Number,
  isPublished: Boolean,
  commentIds: Array,
  author: Object
}
```

  This not only documents your component, but will also warn users in the browser's JavaScript console if they pass the wrong type.

Passing Data to Child Components (with Props)

- Passing Static or Dynamic Props

  So far we've just seen how to pass static props (Strings) but you can actually pass ***any*** type of value to **props**:

Passing a Number

```HTML
<!-- Even though `42` is static, we need v-bind to tell Vue that -->
<!-- this is a JavaScript expression rather than a string.       -->
<blog-post v-bind:likes="42"></blog-post>

<!-- Dynamically assign to the value of a variable. -->
<blog-post v-bind:likes="post.likes"></blog-post>
```

Vue.js

Passing Data to Child Components (with Props)

- Passing Static or Dynamic Props

  So far we've just seen how to pass static props (Strings) but you can actually pass ***any*** type of value to **props**:

  ### # Passing a Boolean

```HTML
<!-- Including the prop with no value will imply `true`. -->
<blog-post is-published></blog-post>

<!-- Even though `false` is static, we need v-bind to tell Vue that -->
<!-- this is a JavaScript expression rather than a string.          -->
<blog-post v-bind:is-published="false"></blog-post>

<!-- Dynamically assign to the value of a variable. -->
<blog-post v-bind:is-published="post.isPublished"></blog-post>
```

Vue.js

Passing Data to Child Components (with Props)

- Passing Static or Dynamic Props

  So far we've just seen how to pass static props (Strings) but you can actually pass ***any*** type of value to **props**:

  ### # Passing an Array

```HTML
<!-- Even though the array is static, we need v-bind to tell Vue that -->
<!-- this is a JavaScript expression rather than a string.            -->
<blog-post v-bind:comment-ids="[234, 266, 273]"></blog-post>

<!-- Dynamically assign to the value of a variable. -->
<blog-post v-bind:comment-ids="post.commentIds"></blog-post>
```

Vue.js

Passing Data to Child Components (with Props)

- Passing Static or Dynamic Props

  So far we've just seen how to pass static props (Strings) but you can actually pass ***any*** type of value to **props**:

  ### # Passing an Object

```HTML
<!-- Even though the object is static, we need v-bind to tell Vue that -->
<!-- this is a JavaScript expression rather than a string.              -->
<blog-post v-bind:author="{ name: 'Veronica', company: 'Veridian Dynamics' }"></b

<!-- Dynamically assign to the value of a variable. -->
<blog-post v-bind:author="post.author"></blog-post>
```

Passing Data to Child Components (with Props)

- Passing Static or Dynamic Props

  So far we've just seen how to pass static props (Strings) but you can actually pass ***any*** type of value to **props**:

Passing the Properties of an Object

If you want to pass all the properties of an object as props, you can use `v-bind` without an argument (`v-bind` instead of `v-bind:prop-name`). For example, given a `post` object:

JS

```
post: {
  id: 1,
  title: 'My Journey with Vue'
}
```

Passing Data to Child Components (with Props)

- Passing Static or Dynamic Props

  So far we've just seen how to pass static props (Strings) but you can actually pass ***any*** type of value to **props**:

The following template:

```HTML
<blog-post v-bind="post"></blog-post>
```

Will be equivalent to:

```HTML
<blog-post
  v-bind:id="post.id"
  v-bind:title="post.title"
></blog-post>
```

Passing Data to Child Components (with Props)

- One-Way Data Flow

  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 prevents child components from accidentally mutating the parent's state, which can make your app's data flow harder to understand.

  In addition, every time the parent component is updated, all props in the child component will be refreshed with the latest value. This means you should **not** attempt to mutate a prop inside a child component. If you do, Vue will warn you in the console.

  There are usually two cases where it's tempting to mutate a prop:

Passing Data to Child Components (with Props)

- One-Way Data Flow

1. **The prop is used to pass in an initial value; the child component wants to use it as a local data property afterwards.** In this case, it's best to define a local data property that uses the prop as its initial value:

```js
props: ['initialCounter'],
data: function () {
  return {
    counter: this.initialCounter
  }
}
```

Vue.js

Passing Data to Child Components (with Props)

- One-Way Data Flow

2. **The prop is passed in as a raw value that needs to be transformed.** In this case, it's best to define a computed property using the prop's value:

```js
props: ['size'],
computed: {
  normalizedSize: function () {
    return this.size.trim().toLowerCase()
  }
}
```

Passing Data to Child Components (with Props)

https://vuejs.org/v2/guide/components-props.html

Case Study

LABS IN ACTION

Analysing our Case Study

So now that we've covered some more detail about Components and passing data to Child Components via **props**, let's take a closer look at how this is implemented in **DonationVue**.

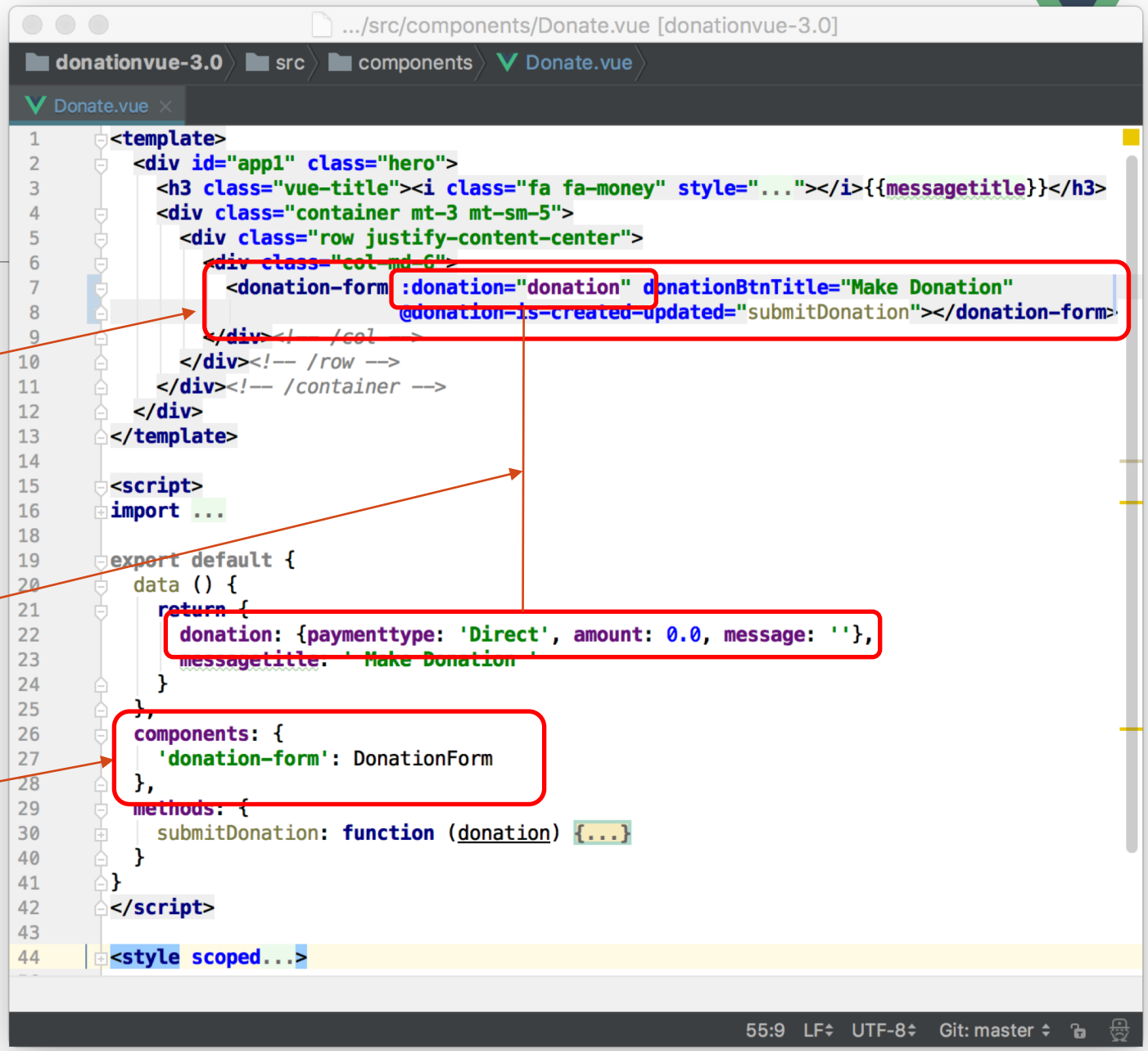
We'll have a brief look at **`Donate.vue`** and **`DonationForm.vue`** with respect to how we can include Custom Components and pass data from Parent to Child to allow for easier maintainability and reusability.

Donate.vue

Child Component

Data Binding (via **props**)

Component Registration

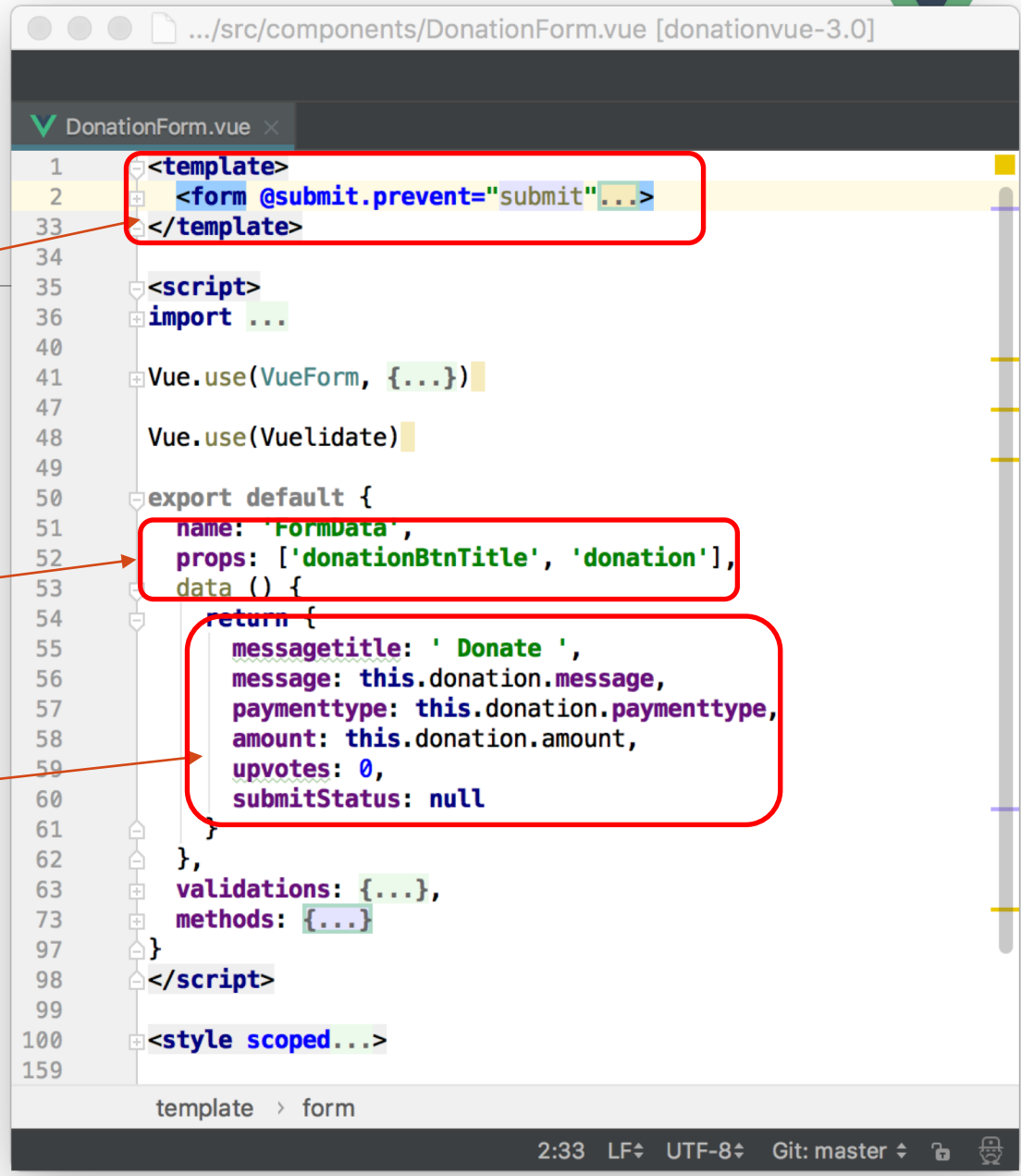


DonationForm.vue

Component Template

Component Props

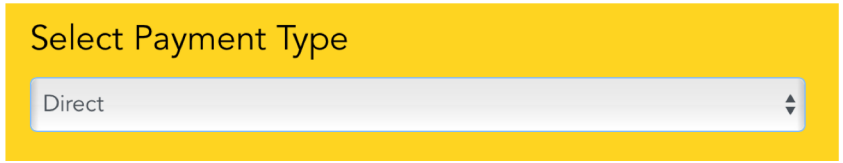
Component Data

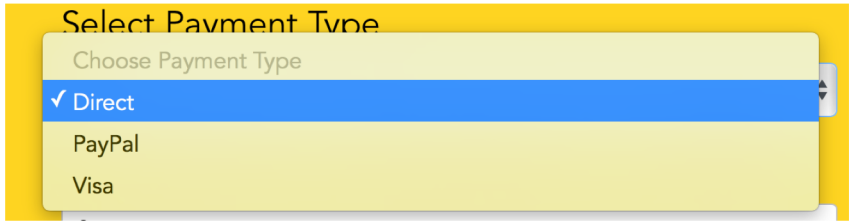


Populated from **props** (because we want to change this **locally**)

DonationForm.vue

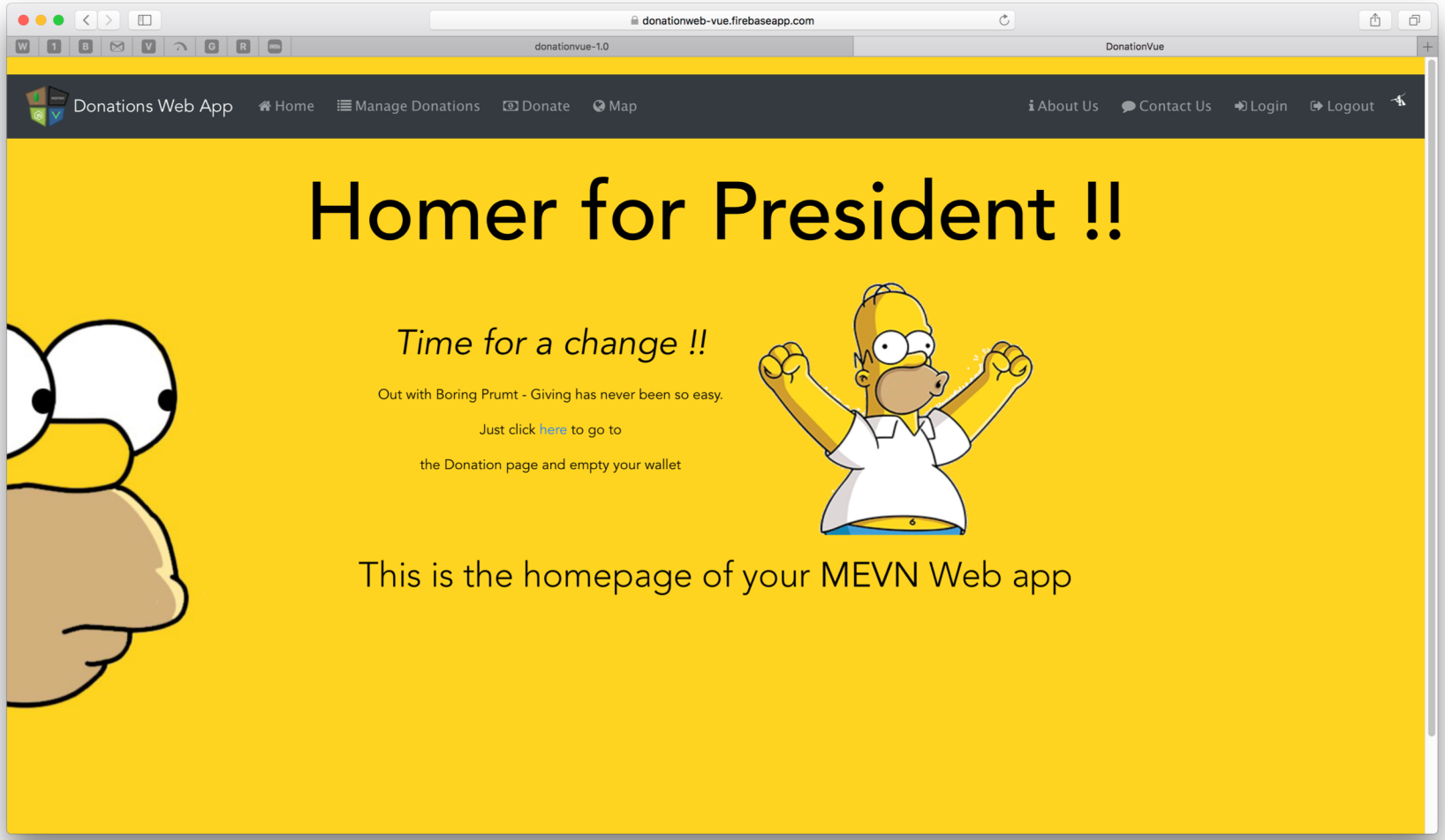
```
<label class="form-label">Select Payment Type</label>
<select id="paymenttype" name="paymenttype" class="form-control" type="text" v-model="paymenttype">
  <option value="null" selected disabled hidden>Choose Payment Type</option>
  <option value="Direct">Direct</option>
  <option value="PayPal">PayPal</option>
  <option value="Visa">Visa</option>
</select>
```





Part of our Input Form

Demo Application https://donationweb-vue.firebaseapp.com



References

- https://vuejs.org

Questions?