DANIEL P. CLARK
FREELANCE DEVELOPER

VueJS as a Frontend for Rails

About the Author.

Daniel P. Clark is a freelance developer, as well as a Ruby and Rust enthusiast. He writes about Ruby on his personal site.

Codeship is a fully customizable hosted Continuous Integration and Delivery platform that helps you build, test, and deploy web applications fast and with confidence.

Learn more about Codeship here.

VueJS as a Frontend for Rails

VueJS is one of the fastest rising stars in the JavaScript frontend ecosystem. It largely embodies simplicity and composability of frontend design solutions without going overboard.

It provides a more elegant way to reduce complexity in both scripting and your styling by grouping them into components. This protects your site's styles from conflicts and also provides logical organization for individual parts of your frontend code.

ABOUT THIS EBOOK

In this eBook we will talk about how VueJS provides an elegant way to reduce JavaScript complexity in both scripting and styling.

Getting Started

Some brief setup instructions.

CODE

```
gem install rails --version "5.2.0.rc1"
rails _5.2.0.rc1_ new vue_example --webpack=vue
cd vue_example
```

From this point, you can start work on VueJS without CoffeeScript support (we'll add that later). Rails includes an example of both frontend VueJS integration and what's called a component.

These files are available at `app/javascript/packs/hello_vue.js` and the component at `app/javascript/hello.vue`. If you would like to challenge yourself to learn the process of integrating these, this is a good place to start.

The Rails Vue Example

You may follow the instructions in this section if you wish to try Rails' small challenge. Comment out the existing code in `hello_vue.js` and uncomment the code in the last section:

CODE

```
import TurbolinksAdapter from 'vue-turbolinks';
import Vue from 'vue/dist/vue.esm'
import App from '../app.vue'

Vue.use(TurbolinksAdapter)

document.addEventListener('turbolinks:load', () => {
  const app = new Vue({
    el: '#hello',
    data: {
      message: "Can you say hello?"
    },
    components: { App }
  })
})
```

Create a route and controller to work with and add the root route to the config to point to it.

CODE

```
rails g controller LandingPage index
```

And add to the `config/routes.rb` file:

CODE

```
root to: "landing_page#index"
```

You can test that this works by running `rails s` and having your web browser load `http://localhost:3000`. From ther, the challenge is up to you to learn what HTML- and JavaScript-related code to put in the site template and landing page to get both VueJS examples to work. That's there for you to do, now let's go and do our own form implementation in VueJS.

Vue JS Rails Form Example

For this example, we're going to create a form for a writer to keep track of their own documents – it will contain a subject, body of text, and the state of revision.

First, let's generate the scaffolding for the document resource.

CODE

```
1  rails g scaffold Document subject:string:index body:text state:integer:index
```

Then edit the migration file under `db/migrate` and change the line for state to provide a default value.

CODE

```
1  t.integer :state, default: 0, null: false
```

Now we can run `rails db:migrate` to update or database. Next we need to update our model for the different states the document may be in. Open up `app/models/document.rb` and add the following:

CODE

```
1  class Document < ApplicationRecord
2    enum state: [:concept, :alpha, :beta, :draft, :publish]
3  end
```

At this point, we're ready to start seeing the changes we'll be making, so first we'll create a CoffeeScript file and

then start a Rails server so we can refresh our browser to work with the results. In a new terminal window, run the following from your project directory:

CODE

```
touch app/javascript/packs/documents.coffee
rails s
```

Now with a browser window open, navigate to `http://localhost:3000/documents`. Here you can use the Rails CRUD for your document resource. We'll be replacing the form to be VueJS-specific.

To start, we'll need to first add the ability to insert our JavaScript pack into our site's header. So open your application template `app/views/layouts/application.html.erb` and add the following between the `<head>` and `</head>` tags.

CODE

```
<% if content_for? :head %>
  <%= yield :head %>
<% end %>
```

Now we have a hook we can use on any page if we use `content_for(:head)` and give it a code block, which will be written to the head section of our specific pages.

Open up `app/views/documents/_form.html.erb` and erase all the contents of the file. This form is used for new entries and updating existing entries in Rails for our documents. First, let's put in the header code block to load what will be our VueJS code.

CODE

```
<% content_for :head do -%>
  <%= javascript_pack_tag 'documents' %>
<% end -%>
```

At this point, trying to load `localhost:3000/documents` in our browser won't work; we need it to recognize our `.coffee` file extension. You can stop the server running in the terminal with CTRL-C and run the following.

CODE

```
bundle exec rails webpacker:install:coffee
```

> Caution: Be sure to do your new feature installations in small steps all while verifying they work before adding more features. Otherwise this, plus a bunch of `yarn add` commands before testing, can lead to this feature not working at all.

Now you can run `rails s` again and bring your browser back to `localhost:3000/documents` and see that the page loads without any errors. We can continue on to the form now. Let's update the same file we were just working on to the following.

CODE

```
<% content_for :head do -%>
  <%= javascript_pack_tag 'documents' %>
<% end -%>

<%= content_tag :div,
  id: "document-form",
  data: {
    document: document.to_json(except: [:created_at, :updated_at])
  } do %>

  <label>Subject</label>
  <input type="text" v-model="document.subject" />

  <label>State</label>
  <select v-model="document.state">
    <%= options_for_select(Document.states.keys, "concept") %>
  </select>

  <label>Body</label>
  <textarea v-model="document.body" rows="20" cols="60"></textarea>

  <br />

  <button v-on:click="Submit">Submit</button>

<% end %>
```

Before writing the CoffeeScript implementation for our VueJS code, let's briefly go over what we have in the file above. The first block of code we've already discussed will load our CoffeeScript asset code in the header through our application template. The `content_tag` will

create a div that stores our current pages' document object as JSON. The document that's created or loaded in the controller gets converted there, and this is what the VueJS code will use.

The `v-model` items are all VueJS-specific names our code will keep track of. For the select `field`, I've found that the Rails `options_for_select` is much easier to work with than VueJS' `v-for` technique, as it's problematic trying to get it to select a `selected` option. And yes, I've tried the half dozen variations of how-tos available on the web for it all to no avail. There is a multi-select add-on you could install with Yarn, but that's a bit excessive for our simple use case.

The `v-on:click` will call the `Submit` function in our Vue object (once we define it) to perform the actions defined there.

Before continuing on, I'd like to share how the basic VueJS `option` implementation would work if we used that here instead.

CODE

```
<select v-model="document.state">
  <option v-for="state in <%= Document.states.keys.to_json %>"
    :value=state
  >
    {{ state }}
  </option>
</select>
```

You should recognize the ERB template `<%= %>`, which will have Ruby get our states of the document and prepare it as JSON. The `v-for` is part of Vue's own DSL, which treats the content like normal for loop code. For every document state, this will duplicate the HTML `<option>` tags and put the replacement word for the state variable on both the value parameter and in the `{{ }}` place.

One last thing I'd like to point out that VueJS has is from their core documentation:

CODE

```
<my-component
  v-for="(item, index) in items"
  v-bind:item="item"
  v-bind:index="index"
  v-bind:key="item.id"
></my>
```

We haven't covered components yet, but what I'd like to point out in this example is that the use of `v-bind` here will execute what's in the quotation marks as regular

JavaScript. So each of these values gets assigned from the JS scope.

Now onto the CoffeeScript VueJS code for our form.

The Code

Now we need to install some Yarn dependencies for having Vue work with Turbolinks in Rails and for more convenient PUT/POST commands.

CODE

```
yarn add vue-resource vue-turbolinks
```

Now our VueJS code can be written in `app/javascript/packs/documents.coffee`. You get extra credit if you've already realized that by using the word 'document' we've used a conflicting JavaScript keyword. Because this is the case, we'll use the variable `ourDocument` in the script to keep things clear and working.

CODE

```
import Vue from 'vue/dist/vue.esm'
import TurbolinksAdapter from 'vue-turbolinks'
import VueResource from 'vue-resource'

Vue.use(VueResource)
Vue.use(TurbolinksAdapter)

document.addEventListener('turbolinks:load', () ->
  Vue.http.headers.common['X-CSRF-Token'] = document
    .querySelector('meta[name="csrf-token"]')
    .getAttribute('content')
```

CODE

```
12
13    if element != null
14      ourDocument = JSON.parse(element.dataset.document)
15
16      app = new Vue(
17        el: element
18
19        data: ->
20          { document: ourDocument }
21
22        methods: Submit: ->
23          if ourDocument.id == null
24            @$http # New action
25              .post '/documents', document: @document
26              .then (response) ->
27                  Turbolinks.visit "/documents/#{response.body.id}"
28                  return
29                (response) ->
30                  @errors = response.data.errors
31                  return
32          else
33            @$http # Edit action
34              .put "/documents/#{document.id}", document: @document
35              .then (response) ->
36                  Turbolinks.visit "/documents/#{response.body.id}"
37                  return
38                (response) ->
39                  @errors = response.data.errors
40                  return
41          return
42    )
43  )
```

The event `turbolinks:load` is the trigger to run this code whenever a page loads in Rails. This code needs to be executed within the `<head>` section of web pages, or you'll get side effects of it not loading without a refresh.

The next line gets the CSRF token, which is needed to verify any data submitted to the server. It takes it from what Rails hands us and assigns it to a response header.

Next we have an assignment of an element with an id of `document-form`. This is an id we've placed in our `content_tag` earlier. The rest of this script is based off of this existing since we do a check `if element != null`.

The `ourDocument` variable is assigned the data we placed in the page as JSON in the `content_tag :div` for the data section. It parses the JSON data, and we continue.

Next we create a Vue object instance in JavaScript (CoffeeScript) with its first parameter being the `element`, which is the document form.

Under `methods`, we have our `Submit` function, which is triggered via the submit button on the page. The `if` conditional that follows that is a check to see if it's a new object and we should use the Rails "new record" path, or if it's an existing object and we'll use PUT to update it.

The `@http`, `post`, `put`, and `then` are all benefits from the `vue-resource` Yarn package we installed earlier. It actually reads out pretty well as is. Just by looking at it, you can see it posts some data to a server URL and then gives us a response. The `response` in parenthesis is a function parameter, and we have two paths for it. The first is the good server response path, and the other is an error situation.

This is surprisingly straight-forward once you know the parts. And with that, we have a VueJS form for our Rails site that works well and loads quickly.

About Components

One of the main attractions about VueJS is its components. What it provides is one location for each component you want to create to have the HTML, JavaScript, and CSS styles all in their own `vue` file. These components boast that the styles won't collide with styles elsewhere on your site. They are well-contained and organized singular functional units of code that may be used most anywhere and can be potentially extended or included within other components. Think of components as the ultimate building block.

If you've done the challenge shown at the beginning of this post, or noticed the component example we breezed by, you most likely have discovered that components get their own XML/HTML tag. The example above is called `<my-component>` and is valid for HTML documents. Doing the Rails provided example will show you just how easy it is to drop a component in place.

Summary

The possibilities with VueJS are pretty high as there are many add-on systems you can integrate with designed to make it work more like a full framework. So you can do as little or as much as you want with it — you're given the liberty to choose however you like.

VueJS has excellent tools to work with for diagnosing both state and issue that may arise. You can get a browser plugin for Chrome or Firefox and even try out their Electron app. Check them out at vue-devtools. Enjoy!

More Codeship Resources.

EBOOKS

Test-Driven Development for JavaScript.

In this eBook we will explore the idea of Test-Driven Development (TDD) for client-side JavaScript.

Download this eBook

EBOOKS

Dockerizing Ruby Apps and Effectively Testing them.

In this eBook you will learn how to dockerize Ruby applications and how to test them.

Download this eBook

EBOOKS

Efficient Project Management for Small Engineering Teams.

In this eBook you will learn how to find balance when managing small engineering teams.

Download this eBook

CODESHIP by CloudBees

Share this Codeship Guide

About Codeship.

Codeship is a hosted Continuous Integration service that fits all your needs. Codeship Basic provides pre-installed dependencies and a simple setup UI that let you incorporate CI and CD in only minutes. Codeship Pro has native Docker support and gives you full control of your CI and CD setup while providing the convenience of a hosted solution.

Codeship Basic

A simple out-of-the-box Continuous Integration service that just works.

Starting at $0/month.


Works out of the box


Preinstalled CI dependencies


Optimized hosted infrastructure


Quick & simple setup

LEARN MORE

Codeship Pro

A fully customizable hosted Continuous Integration service.

Starting at $0/month.


Customizability & Full Autonomy


Local CLI tool


Dedicated single-tenant instances


Deploy anywhere

LEARN MORE