A Killer Vue.js Blog Demo: Launch in 2 Hours Tops

snipcart.com/blog/vuejs-blog-demo

> In a rush? Skip to tutorial or GitHub repo & live demo.

The JS landscape evolves... *briskly*. And that's
putting it mildly.

But amidst the whirlwind of new frameworks & libraries, there ARE awesome tools.

Which ones should you use? Well, a few. My fav right now:

Vue.js

That progressive JavaScript framework is *well-worth* your time (I swear; scout's honor).

Be it to build e-commerce, refactor your app's frontend, craft a complex, SEO-friendly SPA, or launch a simple blog.

And **building a Vue.js blog** is precisely what I'm going to focus on today.

Look, I even made a fancy image for you guys! [photo source]
In this post, I'm going to provide an open source Vue.js blog demo + cover:

- Setup and routing
- Displaying your blog feed with filters
- Displaying individual posts with comments
- Creating a custom Vue plugin to keep your data decoupled

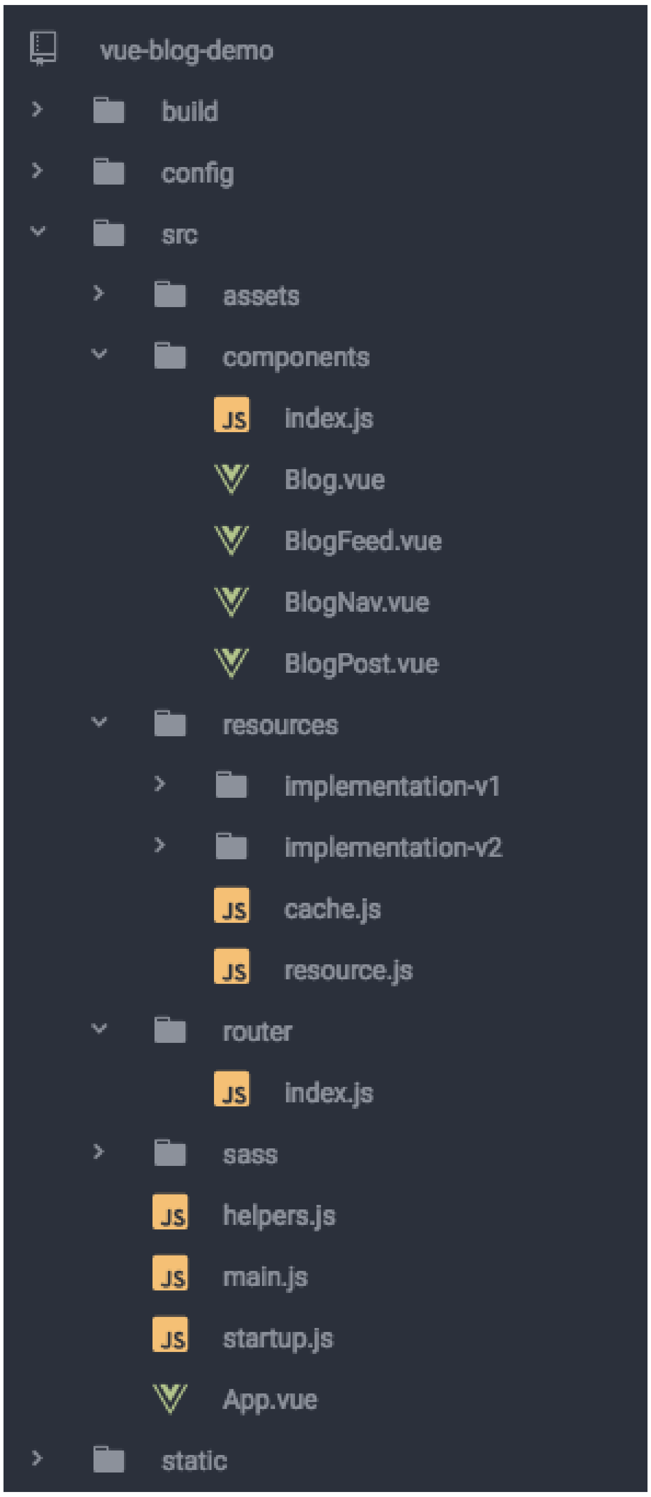
The result will be a JAMstack-ready, truly decoupled Vue blog you can plug to any data source—more on that later.

> Important note: for this post, we assume you have a basic understanding of Vue.js.

Separating concerns in our Vue blog application

Take a quick look at the important bits of the document tree:

I won't go into the details of the webpack setup, but those familiar with Vue might recognize the vue-cli webpack template. We're also using vue-router for... well, routing purposes.



```
vue-blog-demo
  build
  config
  src
    assets
    components
      index.js
      Blog.vue
      BlogFeed.vue
      BlogNav.vue
      BlogPost.vue
    resources
      implementation-v1
      implementation-v2
      cache.js
      resource.js
    router
      index.js
    sass
    helpers.js
    main.js
    startup.js
    App.vue
  static
```

I've only added two folders to the base setup: `src/sass` and `src/resources`. We'll get into why the Vue app's Sass is separate from the components another time.

The `src/resources` folder is where we'll put our decoupled data-access layer which we discuss towards the end of the post.

The component tree itself is nothing groundbreaking:

- `<Blog>` - Our homepage
- `<BlogNav>` - Handles navigation logic
- `<BlogFeed>` - Renders the post listing
- `<BlogPost>` - Renders an individual post

Finally, we've got the router used for URL handling and passing down props.

```
import Vue from 'vue'
import Router from 'vue-router'
import Blog from '../components'

Vue.use(Router)

export default new Router({
  mode: 'history',
  linkActiveClass: 'active',
  routes: [{
    path: '/',
    name: 'feed',
    component: Blog
  }, {
    path: '/by/:author',
    name: 'author',
    props: true,
    component: Blog
  }, {
    path: '/read/:post',
    name: 'post',
    props: true,
    component: Blog
  }]
})
```

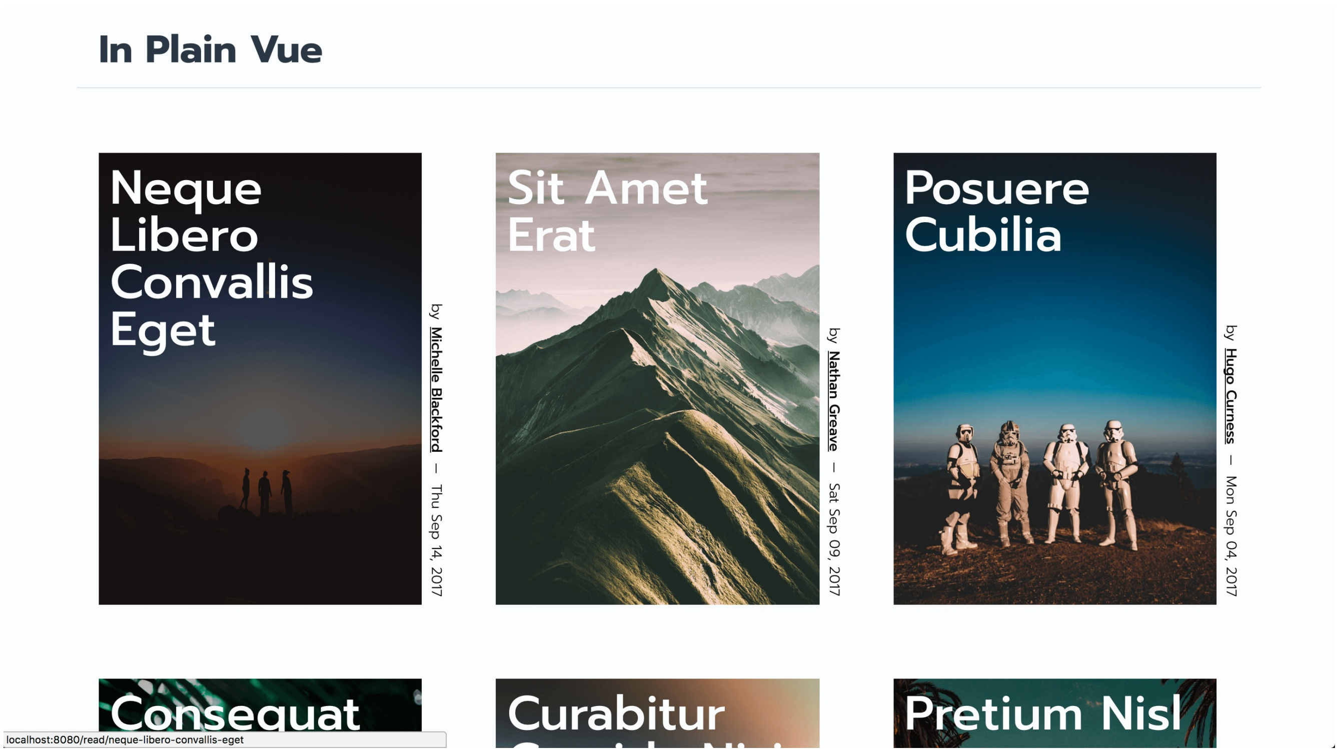
I've taken a slightly non-traditional approach here, as the router will always render the same component. We handle displaying the right content ourselves, the `<Blog>` component being our main hub dispatching props to its three children.

The first route is our site root which just displays our default view (`<BlogFeed>`) unfiltered.

The second is our authors filter, accessed by navigating to `/by/:author`. Vue-Router grabs any path nodes preceded by `:` as variables and injects their value into the route's component as props.

Last but not least, we do the same thing for the `/read/:post` route, which will display a single blog post's content.

Rendering the blog feed


In Plain Vue
Neque Libero Convallis Eget
by Michelle Blackford — Thu Sep 14, 2017
Sit Amet Erat
by Nathan Greave — Sat Sep 09, 2017
Posuere Cubilia
by Hugo Curness — Mon Sep 04, 2017
Consequat
Curabitur
Pretium Nisl
localhost:8080/read/neque-libero-convallis-eget

For now, we'll skip over actually fetching our data and assume it's already been loaded. Here's what the `<BlogFeed>` template looks like:

```
<template>
  <transition-group tag="ul" class="blog__feed" name="preview">
    <li v-for="post in feed" :class="classes" :key="post.id">
      <router-link :to="`/read/${post.id}`">
        <figure class="preview__figure">
          <img :src="post.image"/>

          <transition name="fade">
            <figcaption v-if="!reading" class="preview__title">
                {{ post.title }}
            </figcaption>
          </transition>
        </figure>
      </router-link>

      <transition name="fade">
        <aside v-if="!reading" class="preview__details">
          <h5 class="preview__meta">
            <router-link class="preview__author"
              :to="`/by/${ kebabify(post.author) }`"
              @click.native="scrollTo(0)">
              {{ post.author }}
            </router-link>

            <time class="preview__published">
                {{ prettyDate(post.published) }}
            </time>
          </h5>
        </aside>
      </transition>
    </li>
  </transition-group>
</template>
```

And its logic:

```
import { scrollTo, kebabify, prettyDate } from '../helpers'

export default {
  name: 'blog-feed',
  resource: 'BlogFeed',
  props: { filters: Object },

  data() {
    return { posts: [] }
  },

  computed: {
    reading() { return this.filters.post },
    classes() {
      return {
        'preview': true,
        'blog__post': true,
        'preview--reading': this.reading
      }
    },
    feed() {
      const filterBy = {
        post: (filter, { id }) => filter === id,
        author: (filter, { author }) => filter === this.kebabify(author)
      }

      if (!Object.keys(this.filters).length) return this.posts

      return this.posts.filter(post => {
        return Object.keys(this.filters).every(filter => {
          return filterBy[filter](this.filters[filter], post)
        })
      })
    }
  },

  methods: { scrollTo, kebabify, prettyDate },
  beforeMount() { this.$getResource('feed') }
}
```

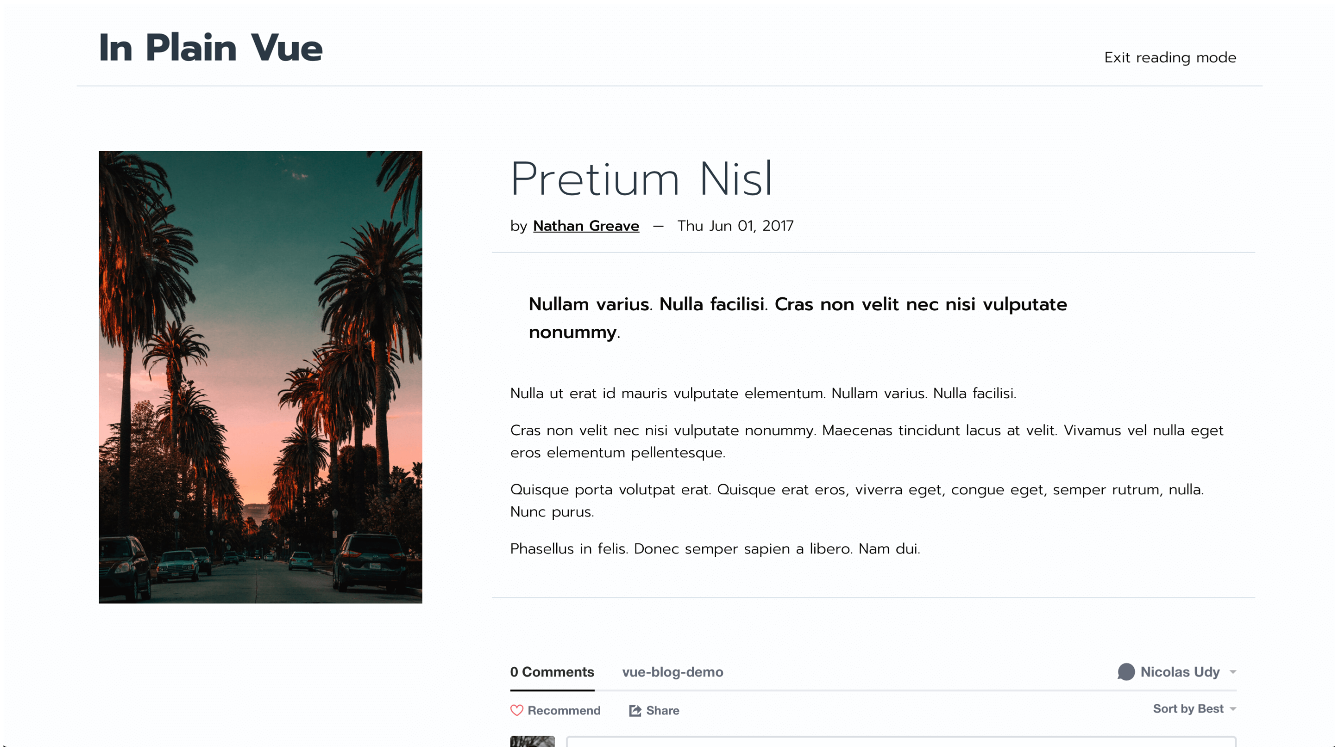
As you can see towards the top of the script, we receive a `filters` object from the parent `<Blog>` component. The `feed()` computed property will take care of automatically handling any changes to the filters as they happen. It filters the post array by looping over each active filter and running the corresponding function against the post, returning only posts that pass every test.

Then, in our template, we just `v-for` the filtered feed, which will keep it up to date at all times. This is probably the most efficient way of handling filters, as you can easily add new ones by appending a new method to `filterBy`.

When a post is clicked, the route will change to that post's ID. However, you may notice that the selected image from the feed remains visible on the left side. We never *actually* hide the listing, we just filter out any post whose ID does not match, leaving us with just one.

Note: in case you're inclined, I recently covered some Vue transitions used here on CSS-Tricks.

Rendering individual blog posts



Pretium Nisl

by Nathan Greave — Thu Jun 01, 2017

Nullam varius. Nulla facilisi. Cras non velit nec nisi vulputate nonummy.

Nulla ut erat id mauris vulputate elementum. Nullam varius. Nulla facilisi.

Cras non velit nec nisi vulputate nonummy. Maecenas tincidunt lacus at velit. Vivamus vel nulla eget eros elementum pellentesque.

Quisque porta volutpat erat. Quisque erat eros, viverra eget, congue eget, semper rutrum, nulla. Nunc purus.

Phasellus in felis. Donec semper sapien a libero. Nam dui.

Okay, we've got the right image displayed on the left side. Now we just need to bring in the corresponding content to its right! Again, it may seem counter-intuitive, but the `<BlogPost>` component is always there, just waiting for something to do.

As soon as a `/read/:post` route is hit, it will load the corresponding post and slide into view using a Vue `<transition>`. The rest is just a plain old Vue template, putting the right variables in place. You'll generally receive the post body with the HTML pre-inserted, so make sure to use the `v-html` attribute instead of `{{ mustaches }}` to avoid auto-escaping tags.

In the demo, I used Disqus along with vue-disqus to add comments to posts. This is what I like the most about the state of frontend development these days: you can add features like this in minutes.

Vue data decoupling: a word on the JAMstack

JAMstack: noun \'jam-stak'\
Modern web development architecture based on client-side JavaScript, reusable APIs, and prebuilt Markup.

The JAMstack (JavaScript, APIs, & Markup) is the product of frontend development's rapid evolution in recent years, notably in the JS community.

> Get up to speed on the JAMstack with this key talk.

IMHO, two of its most redeeming features fuel its rise in popularity:

Ease of access Familiar languages/frameworks coupled with abstracted backends make it hard to resist. I wouldn't be shocked to learn most frontend devs share my dread for databases.

As much as I enjoy working with data, I don't like managing it.

> **Decoupled data source** In theory, it doesn't matter where your data is coming from or how many APIs you're calling to get it. As long as you can feed it to your app. For websites, this means you're never strictly bound to your CMS; you can swap it out if need be!

Why does it matter?

The decoupled backend is without a doubt an attractive prospect. One that I sold to my boss when looking into building a website with Vue.js and a headless CMS. As it turns out, **it's very easy to allow your API's data structure to define your app's inner workings**. Sooner or later, you'll find yourself wondering what happened to the whole "decoupled" argument.

A good hint that you're falling into this trap is if you're fetching data and parsing the response directly in your Vue components. The first step is to remove all your API calls from your components to create a replaceable data-access layer.

There are a ton of tools and techniques to help you implement this sort of pattern. What's difficult is making sure you keep it in mind while building your website or app.

Creating the resource plugin

This demo is a simple, open source example of how you could go about it without adding any dependencies other than `lodash.merge` (4.2kb gzipped).

Often, the simplest way with Vue is to use its plugin system. You've probably made use of it before, like mounting the router or Vuex. Just to refresh our memory: all you need to do is pass a plugin to `Vue.use()` along with any options before creating your root Vue instance.

> Try this post to dive deeper into Vue.js plugins.

Behind the scenes, Vue takes the plugin object and looks for an `install()` method, which it calls passing in Vue as the first argument and your options object as the second.

There a whole bunch of sweet things you can do within this scope. But our mission today is just to create a `$getResource` instance method that you'll be able to call using `this.$getResource(method, options)` from within a component. This approach is pretty cool since it's mounted to each new component instance. That's just a fancy way of saying you get access to the component's `this` binding, just as you're used to.

Head over to `./resources/resource.js` :

```
import _merge from 'lodash.merge'

export default {

  install(Vue, { endpoint = '', resources = {} }) {

    Vue.prototype.$getResource = function(method, options) {

      let name = this.$options.resource

      if (!name || !resources[name] || !resources[name][method]) return;


      let { path, resolve } = resources[name][method](options)


      const mappers = {

        merge: dataSet => {
          _merge(this.$data, dataSet)
          return Promise.resolve(dataSet)
        },


        set: dataSet => {
          Object.keys(dataSet).forEach(prop => {
            this.$set(this.$data, prop, dataSet[prop])
          })

          return Promise.resolve(dataSet)
        }
      }

      // fetch and parse resource then pass it to the resolver
      return fetch(endpoint + path)
        .then(response => response.json())
        .then(response => resolve(response, mappers))
    }
  }
}
```

We assign the `$getResource` method to the Vue prototype. As you can see, our options are `endpoint`, being the API's base URL to which we'll append our query paths, and an object of `resources`, which are "implementations" or definitions indicating how to handle the resource. We'll get into those real soon.

The `install()` method creates a closure, capturing these options in its scope. Any function defined herein will have access to them at all times.

Stepping into the function, after a few checks to make sure we've got everything, we call the resource method, defined as a string by the first argument, and pass in any option received as the second argument. We grab the `path` property and the `resolve` method and define our two mappers: `merge()` and `set()`. The former, using lodash's merge utility, does a deep merge of the `dataSet` with the component's `$data` while the latter loops over `dataSet`'s keys, assigning them to the `$data`.

That last bit is a pretty nice way of adding a shortcut without convoluting your code. Both methods conserve Vue's reactivity and return a Promise, meaning we can chain `.then(dataSet => {})` to run some logic after the fetch is complete.

Finally, the call to ES2015's native `fetch()` method is made. The JSON response is parsed and passed along with the mappers to the `resolve` function.

Resource implementations

With most of the heavy lifting taken care of, we're now ready to look at how we define our resources! If we look at the `./resources` folder in our project you'll see the implementation directory. Open up `implementation/BlogPost.js`:

```
export default {
  post(id) {
    return {
      path: `/post/${id}.json`,
      resolve: (response, mappers) => {
        let { title, content, meta } = response.results[0]
        content = '<p>' + content.split('\n\n').join('</p><p>') + '</p>'

        return mappers.merge({ title, content, ...meta })
      }
    }
  }
}
```

This implementation offers a `post()` method, expecting a post's id to be passed in.

It then returns an object containing the computed path and a resolver function that maps the correct response data to the component. Keep in mind, the goal here is to shape the data to match your components and not the other way around.

Connecting the dots

Now we can get things going by telling Vue to use our custom plugin. First, we create an index in our implementations folder that declares `export { default as ResourceName } from './ResourceName'` for each resource. In `main.js` we pass the plugin and our resource collection to Vue like so:

```
import resource from './resources/resource'
import * as resources from './resources/implementation'

Vue.use(resource, {
    resources,
    endpoint: '/static/api'
})
```

Calling your resources

Now all that's left to do is to call the resources from our components by adding a `resource` option with the name of the resource we want to use.

When we call `this.$getResource` it will know which one to load. I've included some sample JSON data in the `static/api` directory, which is what will be loaded and processed by our plugin.

Take `<BlogPost>` for example.

```
import VueDisqus from 'vue-disqus/VueDisqus'
import { kebabify, prettyDate } from '../helpers'

export default {
  name: 'blog-post',
  resource: 'BlogPost',
  components: { VueDisqus },
  props: { post: String },

  data() {
    return {
      title: '',
      author: '',
      content: '',
      published: '',
      description: ''
    }
  },

  watch: {
    post(to, from) {
      if (to === from || !this.post) return;
      this.$getResource('post', to)
    }
  },

  methods: { kebabify, prettyDate },
  beforeMount() { if (this.post) this.$getResource('post', this.post) }
}
```

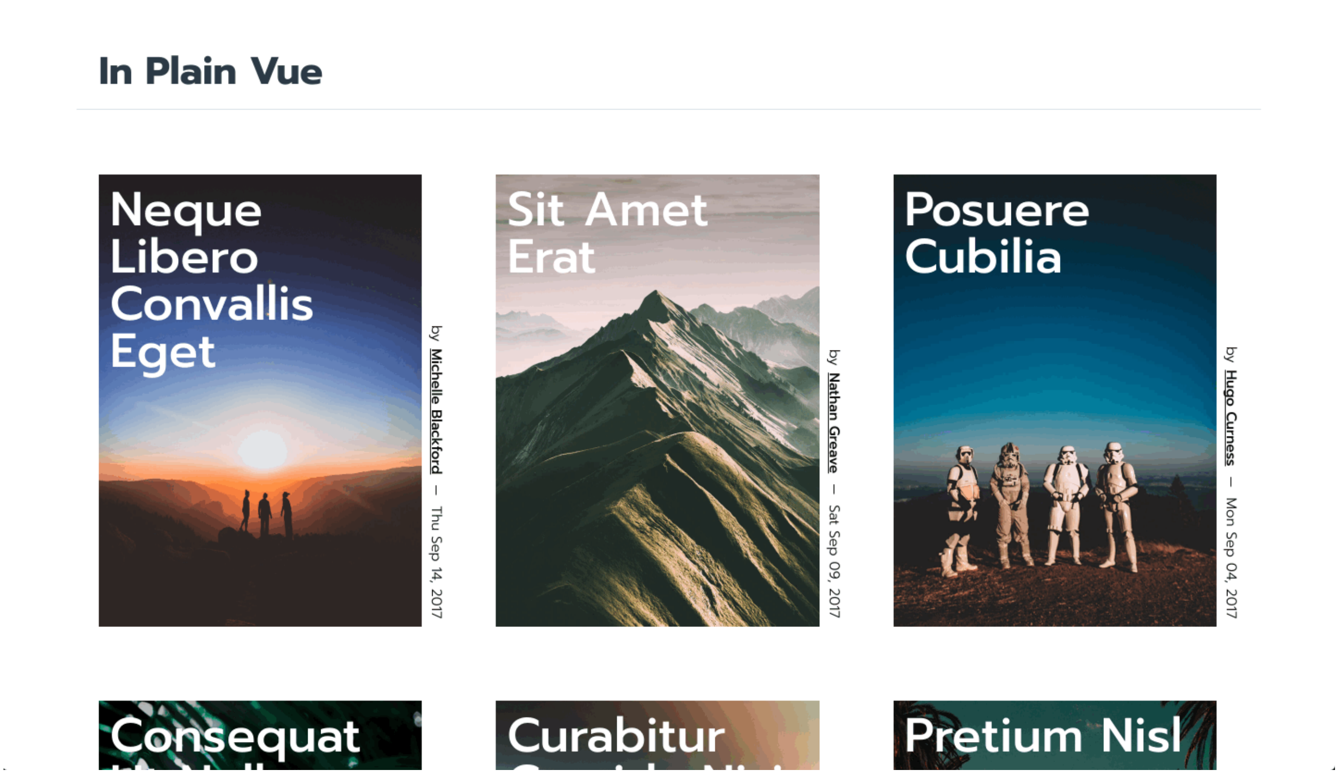
Simple stuff! We've got our component's data object schema, all filled out with placeholders as recommended by Vue and our resource definition. In the `beforeMount()` hook, we check to see if the visitor has landed directly on a blog-post route and call it. Otherwise we wait for the `post` property to change and react by loading the new post.

We don't even need to assign the response to the data; it just gets assigned to our component data by the resolvers! Personally, I quite like this approach because:

1. I like things that make my life easier.
2. I like it when other people can figure out what's going on. Our placeholder data is nicely defined and the calls to `$getResource` hint that we're fetching data.

Awesome! We're avoiding any references specific to our API's implementation inside the component. Instead we call our AJAX methods using an identifier, much like you would with Vuex, Vue's official state management plugin.

Result: live Vue.js blog demo (steal it!)


In Plain Vue
Neque Libero Convallis Eget
by Michelle Blackford — Thu Sep 14, 2017
Sit Amet Erat
by Nathan Greave — Sat Sep 09, 2017
Posuere Cubilia
by Hugo Curness — Mon Sep 04, 2017
Consequat
Curabitur
Pretium Nisl

> [See/steal the demo on GitHub]

> [See live demo deployed on Netlify from GH]

Closing thoughts

You should end up with a simple blog application frontend that is 1) darn smooth and 2) *truly* decoupled from the backend. Which means you could completely change API providers & data structure and nothing would change in your Vue components. All the necessary logic is in one place!

Again, my goal in this post is, among other things, to open your eyes to this potential issue so you don't get caught off guard:

Don't adapt your app to the data, adapt your data to your app.

While the above is a given for many backend devs, it might be new territory for frontend ones.

The approach I've proposed works quite well but lacks some flexibility for large-scale projects. If you're keen on managing App/API relationship more closely, I recommend checking out Vue.js' documentation and tools like Vuex centralized data-store, AJAX libraries like vue-resource or Axios and data-persistence layers like JS-Data.

As we've seen here, Vue.js not only offers blazing fast data-reactivity, it's also very flexible and extendable. I some recent projects, I've developed similar data-access layers that even take care of component data injection and validation as well as defining and generating the input forms for the CMS.

Knowing your tools, their strengths/weaknesses and how to leverage them is the key to becoming a solid developer, something I've hopefully nudged you closer to today.