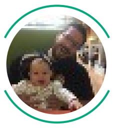
in NuxtJS



Todd Baur Follow
Jul 19, 2018 · 10 min read

I've been using NuxtJS for about a year now and I have a lot of praise to give it. Vue is a great framework: its flexible enough that it only cares about HTML and capable enough to satisfy the larger needs of Fortune 500 companies.

In this article, lets walk through how to quickly spin up a NuxtJS application and how to approach creating a basic RESTful style layout. I'm assuming you've followed Nuxt's instructions for getting started and are staring at the starter page. From this point we'll get busy on making it useful. I'll be using `yarn` to install dependencies, but of course feel free to translate to whatever package manager suites your needs.

About the App

We'll need to add a couple of key dependencies to `package.json` first:

```
...
"dependencies": {
  "@nuxtjs/auth": "^4.5.1",
  "@nuxtjs/axios": "^5.3.1",
  "bootstrap-vue": "^2.0.0-rc.11",
  "lodash.assign": "^4.2.0",
  "lodash.merge": "^4.6.1",
  "nuxt": "^1.0.0",
  "nuxt-i18n": "^3.3.0",
  "vee-validate": "^2.1.0-beta.6",
  "velocity-animate": "^1.5.1",
  "vue-moment": "^4.0.0",
  "vue-notification": "^1.3.10"
},
"devDependencies": {
  "babel-eslint": "^8.2.1",
  "eslint": "^4.15.0",
  "eslint-friendly-formatter": "^3.0.0",
  "eslint-loader": "^1.7.1",
  "eslint-plugin-vue": "^4.0.0",
  "node-sass": "^4.9.0",
  "postcss-preset-env": "^5.1.0",
  "sass-loader": "^7.0.3"
}
...
```

- @nuxtjs/auth for user login
- @nuxtjs/axios for using axios http client
- bootstrap-vue is our UI component toolkit
- always famous lodash helper for handling merging of objects and arrays
- vee-validate for handling form validation
- velocity-animate/vue-notification for excellent toast notifications
- vue-moment for integrating the famous datetime formatting library with vue

Next we run `yarn` to install everything. Onto configuring `nuxt.config.js:`

```
const i18n = require('./config/locales');
module.exports = {
  /*
  ** Headers of the page
  */
  head: {
    title: 'My App',
    meta: [
      {charset: 'utf-8'},
      {name: 'viewport', content: 'width=device-width,
initial-scale=1'},
      {hid: 'description', name: 'description', content: 'My
App'}
    ],
    link: [
      {rel: 'icon', type: 'image/x-icon', href:
'/favicon.ico'}
    ]
  },
  css: [
    '@/assets/scss/app.scss'
  ],
  /*
  ** Customize the progress bar color
  */
  loading: {color: '#3B8070'},
  /*
  ** Build configuration
  */
  build: {
    /*
    ** Run ESLint on save
    */
    extend(config, {isDev, isClient}) {
      const vueLoader = config.module.rules.find((rule) =>
rule.loader === 'vue-loader')
      vueLoader.options.transformToRequire = {
        'img': 'src',
        'image': 'xlink:href',
        'b-img': 'src',
        'b-img-lazy': ['src', 'blank-src'],
        'b-card': 'img-src',
        'b-card-img': 'img-src',
        'b-carousel-slide': 'img-src',
        'b-embed': 'src'
      };
      if (isDev && isClient) {
        config.module.rules.push({
          enforce: 'pre',
          test: /\.(js|vue)$/,
          loader: 'eslint-loader',
          exclude: /(node_modules)/
        })
      }
    }
  },
  modules: [
    ['bootstrap-vue/nuxt', {css: false}],
    ['nuxt-i18n', i18n],
    ['@nuxtjs/axios'],
    ['@nuxtjs/auth']

  ],
  plugins: [
    {src: '~/plugins/vue-notifications', ssr: false},
    {src: '~/plugins/vee-validate'},
    {src: '~/plugins/vue-moment'}
  ],
  router: {
    middleware: ['auth']
  },
  auth: {
    strategies: {
      local: {
        endpoints: {
          login: {url: '/auth/login', method: 'post',
propertyName: 'jwt'},
          logout: {url: '/auth/logout', method: 'post'},
          user: {url: '/auth/user', method: 'get',
propertyName: 'user'}
        }
      }
    }
  },
  axios: {
    /* set API_URL environment variable to configure access
to the API
```

```
      */
      baseURL: process.env.API_URL ||
  'http://localhost:3001/',
      redirectError: {
        401: '/login',
        404: '/notfound'
      }
    }
  }
```

From top to bottom the key additions are:

- Import a `nuxt-i18n` configuration script at the top, and then using it in the module definitions.
- `css` property to handle configuring Bootstrap the way we like using SCSS
- `build` section has `vueLoader` configuration for transforming image paths in nuxtjs for related `bootstrap-vue` components
- `modules` includes the modules for `bootstrap-vue` , `nuxt-i18n` , `axios` , and `auth` .
- `plugins` configuration scripts for notifications, validate, and moment
- `router` added the `auth` middleware
- Last is the configuration section for auth and axios

When you add plugins to Nuxt they’re virtually the same every time. If you run into problems where `document` or `window` is undefined that is because a dependency expects a browser but Nuxt by default does nodejs server side rendering, where these two things don’t exist.

Add some style

Create `~/app/scss/app.scss` :

```
@import ‘~bootstrap/scss/bootstrap’;
```

Make some plugin files

`~/plugins/vue-notifications.js` :

```
import Vue from 'vue';
import Notifications from 'vue-notification'
import velocity from 'velocity-animate'

Vue.use(Notifications, {velocity});
```

`~/plugins/vue-moment.js`:

```
import Vue from 'vue'
import VueMoment from 'vue-moment'

Vue.use(VueMoment);
```

`~/plugins/vee-validate.js`:

```
import Vue from 'vue'
import VeeValidate from 'vee-validate'

Vue.use(VeeValidate, {
  inject: true,
  fieldsBagName: 'veeFields'
});
```

Now we have a little more work to add support for i18n. Create a folder called `config/locales` and in it an `index.js` file:

```
module.exports = {
  locales: [
    {
      code: 'en',
      iso: 'en-US',
      name: 'English',
      file: 'en.json'
    }
  ],
  defaultLocale: 'en',
  seo: true,
  lazy: true,
  detectBrowserLanguage: {
    cookieKey: 'redirected',
    useCookie: true
  },
  langDir: 'config/locales/',
  parsePages: false,
  pages: {},
  vueI18n: {
    fallbackLocale: 'en'
  }
}
```

You'll see its very easy to add a new locale by just duplicating the object in `locales` array and adding the values needed. And no surprised with `en.json` as it is a standard JSON formatted file.

```
  {
    "actions": "Actions",
    "yes": "Yes",
    "no": "No",
    "edit": "Edit",
    "password": "Password",
    "password_confirm": "Confirm Password",
    "login": "Log In",
    "forgot_password": "Forgot Password?",
    "remove": "Remove",
    "destroy_confirm_title": "Please Confirm",
    "destroy_confirm": "Are you sure you want to remove
  this?",
    "logout": "Log Out",
    "back": "Back",
    "save": "Save",
    "update": "Update",
    "forms": {
      "errors": {
        "required": "Please fill this out.",
        "standard": "A server error occurred."
      }
    },
    "cars": {
      "singular": "car",
      "plural": "cars",
      "new": "New Car",
      "edit": "Edit Car"
    }
  }
```

Some people to keep this file flat as possible and then alphabetize it by key. I like to have my entity objects defined and translate my pages from there. It's really whatever style you want to use here. Just be consistent.

That is our app configuration, now lets build stuff.

We'll probably want a more bootstrapped layout first. Open `~/layouts/default.vue` and add this:

```
<template>
  <main>
    <no-ssr>
    <notifications group="alerts"
                   position="bottom right">
      <template slot="body" slot-scope="props">
        <b-alert :show="props.item.duration || 3000"
                 dismissible
                 :variant="props.item.type || 'info'"
                 @dismissed="props.item.timer=0">
          <p>{{props.item.text}}</p>
          <b-progress :variant="props.item.type"
                      striped
                      :animated="true"
                      :max="props.item.duration"
                      :value="props.item.timer"
                      height="4px">
          </b-progress>
        </b-alert>
      </template>
    </notifications>
    </no-ssr>
    <b-row no-gutters class="mb-3">
      <b-col class="bg-dark">
        <b-navbar toggleable="md" type="dark"
variant="dark">
          <b-navbar-toggle target="nav_collapse"></b-navbar-
toggle>
          <b-navbar-brand href="/">My App</b-navbar-brand>
          <b-collapse is-nav id="nav_collapse">
            <b-navbar-nav>
              <b-nav-item class="text-white" :to="'/cars'">
{{$t('new_car')}}</b-nav-item>
              <b-nav-item class="text-white" :to="'/admin'"
v-if="admin">{{$t('admin')}}</b-nav-item>
            </b-navbar-nav>
            <b-navbar-nav class="ml-auto">
              <b-nav-item class="text-white" :to="'login'"
v-if="!$auth.loggedIn">{{$t('login')}}</b-nav-item>
              <b-nav-item class="text-white" @click="logout"
v-if="$auth.loggedIn">{{$t('logout')}}</b-nav-item>
            </b-navbar-nav>
          </b-collapse>
        </b-navbar>
      </b-col>
    </b-row>
    <b-container fluid>
      <nuxt/>
    </b-container>
  </main>
</template>
<script>
  export default {
    methods: {
      logout() {
        this.$auth.logout()
      },
      admin() {
        return this.$auth.loggedIn && this.$auth.user.admin
      }
    }
  }
</script>
```

Ok we have notifications integrated with bootstrap, showing a countdown timer. Pretty cool huh? Next to that is a pretty bland and basic bootstrap navbar. I'm showing that a user has an 'admin' boolean set on it when they login, and that can toggle elements on or off. `nuxt-auth` provides the user object for us when we log in. Toward the bottom you see a `b-container` component holding our special `nuxt` component render tag where our pages will end up.

Edit: Notice that if you configure a plugin with `ssr: false` then you also use the `no-ssr` component in the template. Otherwise you get an error from HMR complaining about the server content not matching the rendered content.

Fetching, Storing, and mutating Data with Vuex

Nuxt makes it super simple to configure modules for vuex. You just name it the same as the entity you want to manage. In this case we're using *cars,* so let's make `~/store/cars.js` :

```
import merge from "lodash.merge";
import assign from 'lodash.assign';

export const state = () => ({
  list: [],
  car: {},
});

export const mutations = {
  set(state, car) {
    state.list = car
  },
  add(state, value) {
    merge(state.list, value)
  },
  remove(state, {car}) {
    state.list.filter(c => car.id !== c.id)
  },
  mergeCars(state, form) {
    assign(state.car, form)
  },
  setCars(state, form) {
    state.car = form
  }
};

export const actions = {
  async get({commit}) {
    await this.$axios.get(`/cars`)
      .then((res) => {
        if (res.status === 200) {
          commit('set', res.data)
        }
      })
  },
  async show({commit}, params) {
    await this.$axios.get(`/cars/${params.car_id}`)
      .then((res) => {
        if (res.status === 200) {
          commit('mergeCars', res.data)
        }
      })
  },
  async set({commit}, cars) {
    await commit('set', cars)
  },
  async form({commit}, form) {
    await commit('mergeCars', form)
  },
  async add({commit}, car) {
    await commit('add', car)
  },
  create({commit}, params) {
    return this.$axios.post(`/cars`, {car: params})
  },
  update({commit}, params) {
    return this.$axios.put(`/cars/${params.id}`, {car:
params})
  },
  delete({commit}, params) {
    return this.$axios.delete(`/cars/${params.id}`)
  }
};
```

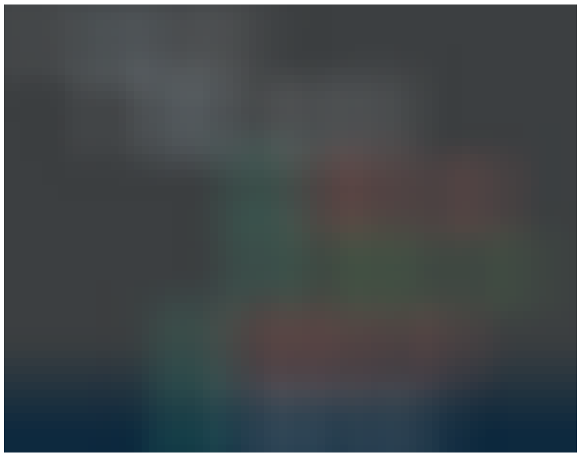
In our `state` we have two keys, one for holding the list of cars, and one for the current car we are looking at.

Mutations are synchronous in vuex, which is important to note. This is organized such that `set, add, remove` all handle mutating the `list` state while `mergeCars, setCars` handle the `car` state.

Actions however do not have to be synchronous. Which makes them a great place to do Ajax calls. We can return promises here, handle mutations, etc. with great ease.

On Nuxt Routing

It took me a bit to wrap my head around how the auto-router worked, so I hope this clears it up for you too.



This folder structure results in the following URLs being generated:

```
/cars -> cars/index.vue (view all)
/cars/:id -> cars/_car_id/index.vue (view one)
/cars/:id/edit -> cars/_car_id/edit.vue (edit view)
/cars/new -> cars/new.vue (new view)
```

When nuxt sees a folder preceded by an underscore, it assumes the folder name is the URL parameter the component needs to receive. You can nest this with new folders too (cars/_car_id/pickups/_id, etc).

`~/cars/index.vue` :

```
<template>
  <b-row>
    <b-col>
      <b-btn variant="outline-success" class="mb-3"
:to="'/cars/new'">
        {{$t('cars.new')}}
      </b-btn>
      <b-table striped hover :items="list" :fields="fields">
        <template slot="name" slot-scope="data">
          <b-link :to="`/cars/${data.item.id}/edit`">
{{data.item.name}}</b-link>
        </template>
        <template slot="members" slot-scope="data">
          {{data.item.users.length}}
        </template>
        <template slot="updated_at" slot-scope="data">
          {{data.item.updated_at | moment("from", "now")}}
        </template>
        <template slot="actions" slot-scope="data">
          <b-btn variant="primary"
:to="'cars/'+data.item.id+'/edit'">
            {{$t('edit')}}
          </b-btn>
          &nbsp;
          <b-btn variant="outline-secondary" @click="id =
data.item.id" v-b-modal.confirmDestroy>
            {{$t('remove')}}
          </b-btn>
        </template>
      </b-table>
      <b-modal id="confirmDestroy"
:title="$t('destroy_confirm_title')" @ok="destroy">
        {{$t('destroy_confirm')}}
      </b-modal>
    </b-col>
  </b-row>
</template>

<script>
  import {mapState} from 'vuex';

  export default {
    async fetch({store}) {
      await store.dispatch('cars/get')
    },
    computed: {
      ...mapState({
        list: state => {
          return state.cars.list
        }
      })
    },
    data() {
      return {
        id: 0,
        fields: [
          {
            key: 'name',
            label: this.$t('name'),
            sortable: true,
          },
          {
            key: 'updated_at',
            sortable: true,
            label: this.$t('updated_at')
          },
          {
            key: 'actions',
            label: this.$t('actions')
          }
        ]
      }
    },
    methods: {
      destroy() {
        this.$store.dispatch('cars/delete', {id:
this.id}).then(() => this.$store.dispatch('cars/get'))
      }
    }
  }
</script>
```

I'm using a little vuex magic here with their `mapState` helper. What this does is literally what it says; it takes the state of cars list and maps it to a local variable named list. The table component in bootstrap-vue is really awesome. With no configuration you get all the fields rendered, but it still exposes itself enough to be super customizable using dynamic template tags. You'll notice I added an 'actions' column with edit and delete buttons. The delete button triggers a confirmation modal. That modal is listening for the `@ok` event, and then calls the `destroy` method when the ok button is clicked.

New and Edit views are virtually identical with only one small change. See if you can spot it:

`~/cars/new.vue` :

```
<template>
    <car-form></car-form>
</template>

<script>
  import CarForm from '~/components/CarForm';
  export default {
    components: {CarForm},
    fetch({store}) {
      store.commit('car/setCar', {})
    }
  }
</script>
```

`~/cars/_car_id/edit.vue` :

```
<template>
  <car-form></car-form>
</template>

<script>
  import CarForm from "~/components/CarForm";
  export default {
    name: "editCar",
    components: {CarForm},
    async fetch({store, params}) {
      await store.dispatch('car/show', {car_id:
params.car_id});
    }
  }
</script>
```

I’m using `fetch` in both cases, but I’m telling vuex in the new method to start with a new `state.cars.car` object. In edit mode, I need to get the current values, so I fetch that from the backend using the params.

This way I only have one form for new and edit:

```
<template>
    <b-form @submit.prevent="submit">
      <b-row align-h="end">
        <b-col class="text-right">
          <b-link :to="'/cars'">
            {{$t('back')}}
          </b-link>
        </b-col>
      </b-row>
      <b-form-group :label="$t('name')">
        <b-form-input id="carNameInput"
                      name="car[name]"
                      ref="name"
                      :state="validateState('name')"
                      type="text"
                      v-validate="{required: true}"
                      data-vv-delay="500"

:placeholder="$t('cars.name_placeholder')"
                      @input="mergeCar({'name': $event})"
                      :value="car.name"></b-form-input>
        <b-form-invalid-feedback>
{{$t('forms.errors.required')}}</b-form-invalid-feedback>
      </b-form-group>
      <b-btn class="my-3" type="submit">{{isUpdate() ?
$t('update') : $t('save')}}</b-btn>
    </b-form>
</template>
```

```
<script>
  import {mapMutations, mapState} from 'vuex';


export default {
    name: "CarForm",
    computed: {
      ...mapState('cars', [
        'car'
      ])
    },
    methods: {
      ...mapMutations('cars', ['mergeCar']),
      isUpdate() { return this.car.hasOwnProperty('id') },
      submit() {
        let vm = this;
        let action = 'cars/' + (this.isUpdate() ? 'update' :
'create');
        this.$validator.validateAll().then((result) => {
          if(result) {
            this.$store.dispatch(action, this.car)
              .then((resp) => {
                let msg = resp.data.name + " " +
(vm.$t('car.' + this.isUpdate() ? 'updated' : 'created'));
                vm.$notify({text: msg, type: 'success',
group: 'alerts'});
                vm.$router.push('/cars')
              })
              .catch((resp) => {
                vm.$notify({text:
vm.$t('forms.errors.standard'), type: 'warning', group:
'alerts'})
              })
          } else {
            vm.$notify({text: vm.$t('forms.errors.invalid'),
type: 'warning', group: 'alerts'})
          }
        })
      },
      validateState(ref) {
        if (this.veeFields[ref] &&
this.veeFields[ref].dirty) {
          return !this.errors.has(ref)
        } else {
          return null
        }
      }
    }
  }
</script>
```

You'll notice i am not using v-model. While I would love to do that, the support for it isn't there with vuex. You can't directly mutate state in strict mode, and undoing strict mode is kind of a bad idea.

The last view is the `show` view. This is the 'read only' version of edit basically. It's a blank canvas in this tutorial really.

```
<template>
  <div>
    <h3>{{$store.state.cars.car.title}}</h3>
  </div>
</template>

<script>
  export default {
    name: "showCar",
    async fetch({store, params}) {
      await store.dispatch('car/show', {car_id:
params.car_id});
    }
  }
</script>

<style scoped>

</style>
```

Pretty much we've taken the same method to fill the store using `fetch` and then we're spitting out what is in the state to the view.

Congrats, you've built a full CRUD MVC dohickey for cars with one attribute! I hope you found this useful and that your imagination can run wild from here. Feel free to comment below if you need help. You can also find me hanging out in the Nuxt gitter, or bootstrap-vue slack rooms if you need help.

Edit: Lots of great feedback and the biggest request was *"where is the code?"*

Here it is: https://github.com/toadkicker/nuxt-crud-example