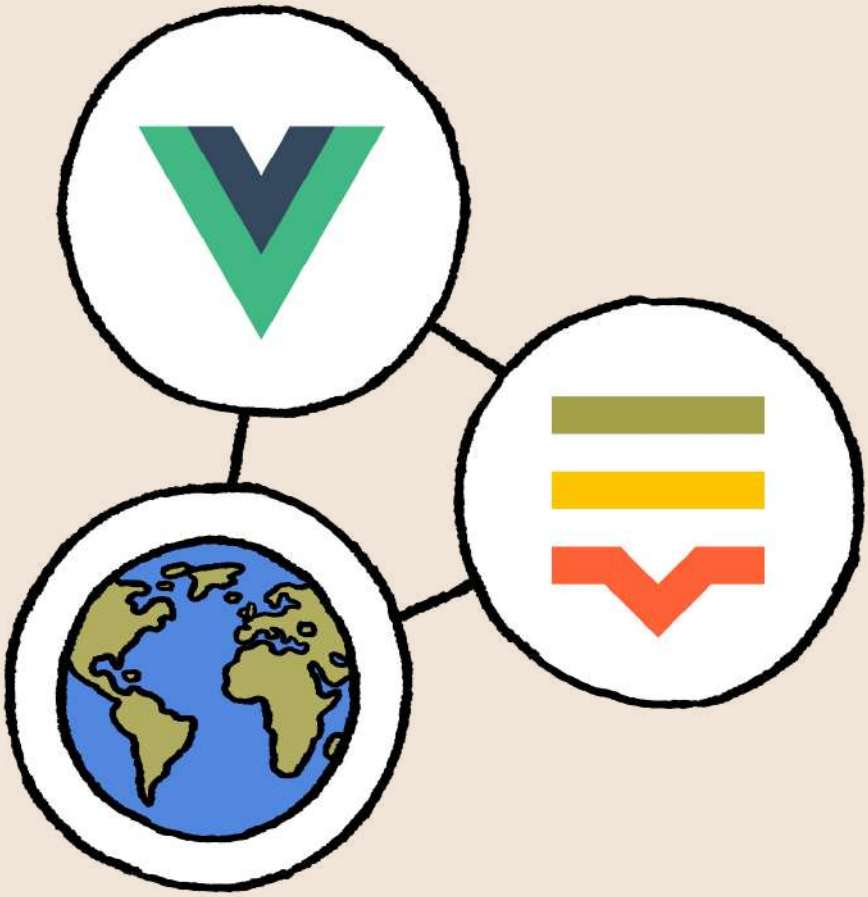
lokalise



Vue internationalization: A simple guide to building a multilingual app

If you're a Vue developer just getting started with internationalization (i18n) – you're in the right place. Read this step-by-step tutorial and learn how to set up a Vue.js website with i18n support to build a multilingual app.

Internationalization is an important yet often missed step in software development. Setting up a Vue.js website with internationalization (i18n) support sounds daunting at first, but it's actually easier than one might think. For this tutorial we will be using Vue I18n, a great package from the core Vue devs.

Loosely based on the original article by Dobromir Hristov. Used with author's permission.

In this article we are going to cover the following topics:

- Installing and setting up Vue I18n
- Adding support for multiple languages
- Storing and using translations
- Implementing pluralization
- Datetime and number localization
- Switching the locale
- Integrating Vue Router and making it work with multiple locales
- Lazy load translation files based on the chosen locale
- Read user preferred locale and adjust the default language based on it

The source code is available on GitHub.

A working demo can be found at: lokalise-vue-demo.herokuapp.com.

Table of contents

Initial Setup of the Vue.js App

In this Vue I18n tutorial we are going to utilize a special CLI so be sure to install it as follows:

```
npm install -g @vue/cli
```

Next, create a new demo project named `vue-i18n-demo`:

```
vue create vue-i18n-demo
```

I've chosen to use the default preset for this app (Babel and ESLint).

Lastly, `cd` into the app folder and install the Vue I18n plugin:

```
cd vue-i18n-demo
vue add i18n
```

You will be asked some questions regarding the project setup. I've provided the following answers:

- The locale of project localization — `en`
- The fallback locale of project localization — `ru` (you may choose any other language)
- The directory where store localization messages of project — `locales`
- Enable locale messages in Single file components — `N`

After installing Vue I18n, the following operations will be performed for you:

1. Create a `vue.config.js` file in the project's root. This file contains general settings for the I18n plugin (default language, fallback, etc.).
2. Create an `.env` file in the project's root. We will return to this file later.
3. Create an `src/i18n.js` file. This file actually imports the necessary I18n modules and does the initial setup.
4. Tweak `src/main.js` to add I18n functionality to the app.
5. Create `src/locales` directory with `en.json` and `ru.json` files which will store our translations.

Now the setup is done, and we can proceed to the next step!

Setting Up Lokalise

Managing translations for multiple languages is a pretty complex task, especially for larger sites. Things become even more complex if you have limited proficiency in the supported languages. What should you do in such cases? Use a proper translation management system!

Meet Lokalise, the leading translation management service which allows you to easily manage translation files, collaborate with translators, order professional translations, enable integrations with various services, and much more. Throughout this article I will briefly explain how to get started with Lokalise. For now, you will need to perform the following steps:

- Grab your free trial (no credit card is required).
- Proceed to Personal profile > API tokens.
- Generate a new read-write token (make sure to keep it safe).
- Download CLIv2 and unpack it somewhere on your PC.

- `cd` into the created directory

- Create a new Lokalise project by running `lokalise2 project create --name VueJS --token YOUR_TOKEN_HERE --languages “[{\”lang_iso\”:\”ru\”},{\”lang_iso\”:\”en\”}]” --base-lang-iso “en”`. Adjust the supported languages as necessary.

- The above command is going to return an object with the project details. Copy the `project_id` as we will need it later.

The basic setup is done, and we can proceed to the next part!

Performing Simple Translations

Storing Vue Translations in JSON Files

To get started with Vue I18n, let's see how to perform basic translations. For instance, we may want to greet our users on the main page of the site. To achieve that, add English translation to `locales/en.json`:

```
{
  "main": {
    "welcome": "Hello from {company}!"
  }
}
```

There are a few things to note here:

- Your translations are stored in simple JSON files. The key will be used in your code and automatically substituted with a proper value based on the chosen language.
- Keys can be nested (this is especially useful for larger sites with many translations).
- Keys support interpolation: take a look at the `{company}` part in the example above. This placeholder will be replaced with an actual company name that we are going to provide in the code. If you are a fan of Ruby on Rails (like me), then you may also use Rails-style placeholders: `%{company}`.

Now add Russian translations to the `locales/ru.json` file:

```
{
  “main”: {
    “welcome”: “Вас приветствует {company}!”
  }
}
```

Great! The translations are ready, and we can use them in our app.

Uploading Translation Files to Lokalise

After creating the initial translation files, we can upload them to the Lokalise project we created earlier. To achieve this, run the following commands:

```
lokalise2 file upload --lang-iso en --file
“PATH_TO_PROJECT\src\locales\en.json” --project-id
PROJECT_ID --token YOUR_TOKEN
lokalise2 file upload --lang-iso ru --file
“PATH_TO_PROJECT\src\locales\ru.json” --project-id
PROJECT_ID  --token YOUR_TOKEN
```

`PROJECT_ID` is the identifier that you received in the previous step when creating a new project.

Using Vue I18n in Templates

Now let’s see how to utilize the created translations in our pages. In fact, all you need to do is utilize a $t function and pass a translation key to it. Let’s tweak the template within the App.vue file:

```
<template>
  <div id=”app”>
```

```
    <p>{{ $t('main.welcome', {company: 'Lokalise'}) }}
</p>
  </div>
</template>
```

`main.welcome` is the translation key. We are using dot (.) because the key is nested. The `company: 'Lokalise'` part provides the value for the `{company}` placeholder within the translation file.

We are not going to use the default `HelloWorld` component, so remove the following line from the `App.vue` file:

```
import HelloWorld from './components/HelloWorld.vue'
```

Also, remove `HelloWorld` from the `components` property and delete the `components/HelloWorld.vue` file.

Now you may boot the app:

```
npm run serve
```

Proceed to `localhost:8080` — the welcoming message should be displayed for you. In order to make sure that the Russian translation works as well, you can open the `i18n.js` file and set `'ru'` as the `locale` property:

```
export default new VueI18n({
  locale: 'ru', // <-------
  fallbackLocale:
process.env.VUE_APP_I18N_FALLBACK_LOCALE || 'en',
  messages: loadLocaleMessages()
})
```

Don’t forget to undo the above change once you’re done testing!

Downloading Translation Files from Lokalise

Once you have edited your translations on Lokalise, download them back into your project by running:

```
lokalise2 file download --unzip-to
PROJECT_PATH\src\locales --format json --token
YOUR_TOKEN --project-id PROJECT_ID
```

Changing the Language

So, our application is now multilingual, but there's no option to actually switch between locales via the user interface. Let's add this feature now!

I suggest creating a new component called `LocaleSwitcher` which is going to handle all switching logic. Create a new `src/components/LocaleSwitcher.vue` file:

```
<template>
<ul>
  <li v-for="locale in locales" :key="locale"
@click="switchLocale(locale)">
    {{locale}}
  </li>
</ul>
</template>
```

This is a basic unordered list which is going to contain our supported locales. Once the list item is selected, we call up the `switchLocale()` method and perform the actual change.

The next step is to provide an array of supported locales. We can just hard-code them, but that is not very convenient. Instead, let's take advantage of the `.env` file which I mentioned earlier. Add the following line:

```
VUE_APP_I18N_SUPPORTED_LOCALE=en,ru
```

Now use this environment variable to produce an array of locales within the LocaleSwitcher.vue:

```
<!-- your template here... -->
```

```
<script>
export default {
  name: 'LocaleSwitcher',
  data() {
    return {
      locales:
process.env.VUE_APP_I18N_SUPPORTED_LOCALE.split(',')
    }
  }
}
</script>
```

`process.env` enables us to access environment variables. Then we just take the required value and use `split()` to produce an array.

Lastly, add a `switchLocale()` method:

```
<script>
export default {
  name: 'LocaleSwitcher',
  methods: { // <--------------------------
    switchLocale(locale) {
      if (this.$i18n.locale !== locale) {
        this.$i18n.locale = locale;
      }
    }
  },
  data() {
    return {
      locales:
process.env.VUE_APP_I18N_SUPPORTED_LOCALE.split(',')
    }
  }
}
</script>
```

`this.$i18n.locale` allows us to get and set the current locale of the app. We first check if the requested locale is different from the currently chosen one and update it if necessary.

To make the list elements look more like links, you may add the following `scoped` style to the component:

```
<style scoped>
li {
  text-decoration: underline;
  color: #459CE7;
  cursor: pointer;
}
</style>
```

Once you are ready, use the new component within `App.vue`:

```
<template>
  <div id="app">
    <LocaleSwitcher /> <!-- <------------------- -->
    <p>{{ $t('main.welcome', {company: 'Lokalise'}) }}
</p>
  </div>
</template>

<script>
import LocaleSwitcher from
'./components/LocaleSwitcher.vue' // <-------------

export default {
  name: 'App',
  components: {
    LocaleSwitcher // <-------------
  }
}
</script>
```

Reload the application, proceed to `localhost:8080` and try clicking the switcher links. The welcoming message should be translated to the chosen language. Good job!

Exploring Vue I18n Features

Pluralization

One of the most common internationalization tasks is pluralizing the text properly. To demonstrate it in action, let's show how many new messages the user has. Start with the English translation and provide the `new_message` key:

```
{
  “main”: {
    “welcome”: “Hello from {company}!”,
    “new_message”: “no messages | one message | {count} messages”
  }
}
```

Here we provide three possible translations separated with pipeline (`|`) which will be picked automatically, based on the passed number. The first translation (`no messages`) is optional and may be omitted.

Pluralization rules in English are quite simple as the word may have only two forms. For Russian, things become more complex:

```
{
  “main”: {
    “welcome”: “Вас приветствует {company}!”,
    “new_message”: “нет сообщений | {count} сообщение | {count} сообщения
| {count} сообщений”
  }
}
```

Unfortunately, Vue I18n does not support complex pluralization out of the box. It is quite simple to add this feature though. Modify the `i18n.js` file like this:

```
import Vue from 'vue'
import VueI18n from 'vue-i18n'

const defaultImpl = VueI18n.prototype.getChoiceIndex // <------------
VueI18n.prototype.getChoiceIndex = function(choice, choicesLength) { //
<------------ Add support for Russian pluralization
  // this === VueI18n instance, so the locale property also exists here
  if (this.locale !== 'ru') {
    return defaultImpl.apply(this, arguments) // default implementation
  }

  if (choice === 0) {
    return 0;
  }

  const teen = choice > 10 && choice < 20;
  const endsWithOne = choice % 10 === 1;

  if (!teen && endsWithOne) {
    return 1;
  }

  if (!teen && choice % 10 >= 2 && choice % 10 <= 4) {
    return 2;
  }

  return (choicesLength < 4) ? 2 : 3;
}

Vue.use(VueI18n)

// other code goes here...
```

Basically, we are explaining what translation to choose based on the provided number. Pluralization algorithms for all languages may be found on the Unicode CLDR website.

With the above code in place, we may implement pluralization in the code. Add a new line to the `App.vue` file:

```
<template>
  <div id="app">
    <LocaleSwitcher />
    <p>{{ $t('main.welcome', {company: 'Lokalise'}) }}</p>
    <p>{{ $tc('main.new_message', 10) }}</p> <!-- <-------- add this -->
  </div>
</template>
```

Note that in order to employ pluralization, you need to use a `$tc` function. It accepts the following three arguments:

- Translation key.
- The number to use in pluralization.
- Optional object with the interpolation values. By default, the second argument will be provided as an interpolation value, but you may override it. For example: `$tc('main.new_message', 10, {count: "Not many"})`.

Date and Time

The next common feature that I would like to show you is localization of date and time. In order to support it, we need to provide one or more datetime formats within the i18n.js:

```
// other code goes here...

const dateTimeFormats = {
  'en': {
    short: {
      year: 'numeric',
      month: 'short',
      day: 'numeric'
```

```
    }
  },
  'ru': {
    short: {
      year: 'numeric',
      month: 'short',
      day: 'numeric'
    }
  }
}

export default new VueI18n({
  locale: process.env.VUE_APP_I18N_LOCALE || 'en',
  fallbackLocale:
process.env.VUE_APP_I18N_FALLBACK_LOCALE || 'en',
  messages: loadLocaleMessages(),
  dateTimeFormats // <--------- make sure to add this line as well!
})
```

For both locales we have added a new format called `short`:

- The year and day will be provided as numbers.
- The name of the month will be displayed in a shortened form (for example, `Apr`).

You can add other formats as necessary by using the options listed in ECMA-402.

Now, let's localize the current date using the `short` format. Add a new line to the `App.vue` file as follows:

```
<template>
  <div id="app">
    <LocaleSwitcher />
    <p>{{ $t('main.welcome', {company: 'Lokalise'}) }}</p>
    <p>{{ $tc('main.new_message', 10) }}</p>
    <p>{{ $d(new Date(), 'short') }}</p> <!-- <----------------- -->
  </div>
</template>
```

`$d` is a function that will perform localization. `new Date()` returns the current date, whereas `short` is the name of our format. Sweet!

Working with Currency

The last Vue I18n feature that I am going to show you is formatting numbers and representing them with the proper currencies. To add currency formats, we need to tweak the `i18n.js` file again:

```
// other code goes here...

const numberFormats = {
  'en': {
    currency: {
      style: 'currency', currency: 'USD'
    }
  },
  'ru': {
    currency: {
      style: 'currency', currency: 'RUB'
    }
  }
}

export default new VueI18n({
  locale: process.env.VUE_APP_I18N_LOCALE || 'en',
  fallbackLocale: process.env.VUE_APP_I18N_FALLBACK_LOCALE || 'en',
  messages: loadLocaleMessages(),
  dateTimeFormats,
  numberFormats // <------ Make sure to add this line!
})
```

So, for the English locale we'll be using US dollars, whereas for Russian we'll use roubles.

Next, add another new line to the `App.vue` file:

```
<template>
  <div id="app">
    <LocaleSwitcher />
    <p>{{ $t('main.welcome', {company: 'Lokalise'}) }}</p>
    <p>{{ $tc('main.new_message', 10) }}</p>
    <p>{{ $d(new Date(), 'short') }}</p>
    <p>{{ $n(100, 'currency') }}</p> <!-- <------------ -->
  </div>
</template>
```

$n is a function for localizing numbers. It accepts the actual number and the name of the format.

Vue I18n docs provide some more examples explaining how to add custom formatting for your numbers.

Setting the Language as a Route Parameter

At this point, our application is translated into two languages. However, we do not persist the chosen locale anywhere. Also, whenever a user visits our site, an English locale is being set by default. In this section we'll see how to enhance our application further with the help of Vue Router.

Installing Vue Router

The first step is installing Vue Router itself:

```
vue add router
```

During the installation, choose to use history mode.

The above command is going to perform the following operations:

1. Tweak `main.js` to enable routing.
2. Add sample router links and a `router-view` to the `App.vue`.
3. Create a `router/index.js` file with initial configuration.
4. Add two sample views (within the `views` folder).

Managing Views

What I would like to do next is tweak the `App.vue` file and update the sample views created for us in the previous section. Start by tweaking the `App.vue`. We are going to bring the language switcher back as well as introduce a new `MainMenu` component:

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

<script>
import LocaleSwitcher from
'@/components/LocaleSwitcher.vue'
import MainMenu from '@/components/MainMenu.vue'

export default {
  name: 'App',
  components: {
    LocaleSwitcher,
    MainMenu
  }
}
</script>
```

Next, create a new `components/MainMenu.vue` file which is going to contain the router links generated for us during router installation:

```
<template>
<div id="nav">
  <router-link to="/">{{ $t('menu.home') }}</router-link> |
  <router-link to="/about">{{ $t('menu.about') }}</router-link>
</div>
</template>
```

Add new translations for the links. English:

```
{
  "menu": { // <------------
    "home": "Home",
    "about": "About"
```

```
 },
  “main”: {
    “welcome”: “Hello from {company}!”,
    “new_message”: “no messages | one message | {count} messages”
  },
  “about”: {
    “welcome”: “Welcome to the About page!”
  }
}
```

And Russian:

```
{
  “menu”: { // <---------------
    “home”: “Главная”,
    “about”: “О нас”
  },
  “main”: {
    “welcome”: “Вас приветствует {company}!”,
    “new_message”: “нет сообщений | {count} сообщение | {count} сообщения
| {count} сообщений”
  },
  “about”: {
    “welcome”: “Добро пожаловать на страницу О нас!”
  }
}
```

Now we need to tweak the sample views generated for us. The `views/Home.vue` will contain the translated messages which were displayed on the main page:

```
<template>
<div class=”home”>
  <p>{{ $t(‘main.welcome’, {company: ‘Lokalise’}) }}</p>
  <p>{{ $tc(‘main.new_message’, 10) }}</p>
  <p>{{ $d(new Date(), ‘short’) }}</p>
  <p>{{ $n(100, ‘currency’) }}</p>
</div>
</template>
```

The `views/About.vue` will simply display a welcoming message:

```
<template>
<div class=”about”>
  <p>{{ $t(‘about.welcome’) }}</p>
</div>
</template>
```

Provide translations for the `about.welcome` key:

```
“about”: {
  “welcome”: “Welcome to the About page!”
}
```

```
“about”: {
  “welcome”: “Добро пожаловать на страницу О нас!”
}
```

Adding Routes with Vue I18n Support

The views are ready, and we can proceed to our routes. Replace the contents of the `router/index.js` file with the following:

```
import Vue from ‘vue’
import VueRouter from ‘vue-router’
import i18n from ‘../i18n’

function load(component) {
 // ‘@’ is aliased to src/components
 return () => import(`@/views/${component}.vue`)
}
```

```
Vue.use(VueRouter)
```

`load()` is a small helper function that we will use to load the proper view based on the current router.

Now add the actual routes as follows:

```
// other code ...

const routes = [{
    path: '/:locale',
    component: {
      template: "<router-view></router-view>"
    },
    children: [{
        path: '',
        name: 'Home',
        component: load('Home')
      },
      {
        path: 'about',
        name: 'About',
        component: load('About')
      }
    ]
  }
]
```

`/:locale` is a variable part of the route which will equal either `en` or `ru`. For example: `localhost:8080/en/about`. There are also two child routes for our Home and About views. Home is the default page; therefore, its path is set to an empty string (it can be accessed with `locahost:8080/en`).

When the `/:locale` page is initially entered, we need to set a proper locale based on the parameter. Therefore, add a `beforeEnter` property:

```
const routes = [{
    path: '/:locale',
    component: {
      template: "<router-view></router-view>"
    },
    beforeEnter: (to, from, next) => { // <------------
      const locale = to.params.locale; // 1
      const supported_locales = process.env.VUE_APP_I18N_SUPPORTED_LOCALE.
split(','); // 2
      if (!supported_locales.includes(locale)) return next('en'); // 3

      if (i18n.locale !== locale) { // 4
        i18n.locale = locale;
      }

      return next() // 5
    },
    children: [
      // children here...
    ]
  }
]
```

The logic is pretty simple:

1. We fetch the requested locale from the route by using `to.params.locale`.
2. We also generate an array of supported locales.
3. If the requested locale is not supported (that is, not found in the array), we set the locale to English and redirect the user to `localhost:8080/en`.
4. If the locale is supported and the current locale is different, we switch to the chosen language.
5. Lastly, navigate the user to the page.

We may also handle all other routes and redirect the user to the default locale:

```
const routes = [{
    path: '/:locale'
    // other code for the /:locale route
```

```
  },
  { // <---------------
    path: '*',
    redirect() {
     return process.env.VUE_APP_I18N_LOCALE;
    }
  }
]
```

Lastly, create the router:

```
// routes and other code ...

const router = new VueRouter({
  mode: 'history',
  base: process.env.BASE_URL,
  routes
})

export default router
```

The routes are now configured properly!

Adjusting the Navigational Links

The links within the `components/MainMenu.vue` file also require some tweaks:

```
<template>
<div id="nav">
  <router-link :to="{ name: 'Home', params: { lang: this.$i18n.locale }}">
    {{ $t('menu.home') }}
  </router-link>

  <br>

  <router-link :to="{ name: 'About', params: { lang: this.$i18n.locale
```

```
}}">
    {{ $t('menu.about') }}
  </router-link>
```

Here we utilize named routes (`Home` and `About` respectively). These named routes were set in the previous step using the following line: `name: 'Home'`. For each route we also provide the currently set locale.

If you boot the server now, you will probably get an error saying that the template compiler is not available. To fix it, add the following property to the `vue.config.js` file:

```
module.exports = {
  runtimeCompiler: true, // <-------
  pluginOptions: {
    // other options...
  }
}
```

The navigational links and language switcher should work now. One problem, however, is that after changing a language, the locale is not being updated in the URL. Let's fix that as well!

Rewriting URL After Locale Switch

Updating a URL after a locale switch is actually a very simple thing to do. We need to add a single line to the `components/LocaleSwitcher.vue` file:

```
// your template...

<script>
export default {
  name: 'LocaleSwitcher',
```

```
  methods: {
    switchLocale(locale) {
      if (this.$i18n.locale !== locale) {
this.$i18n.locale = locale;
        const to = this.$router.resolve({ params: {locale} }) // <--------
------------
        this.$router.push(to.location) // <--------------------
      }
    }
  },
  data() {
    // data...
  }
}
</script>
```

First, we are generating the proper URL based on the chosen locale. Then, `this.$router.push` will navigate the user to a new URL (it is possible to return to the previous one thanks to the history mode).

Now if you change the locale, the URL will be updated for you. Try playing around with the links to make sure everything works properly!

Introducing Lazy Loading

Our application is working but there is one important feature missing: lazy loading of translation files. Usually, loading translations for all languages is an overkill. Therefore, in this section we will see how to implement this feature. Our application is working but there is one important feature missing: lazy loading of translation files. Usually, loading translations for all languages in an overkill. Therefore, in this section we will see how to implement this feature.

Introducting Translation Plugin

Before implementing lazy loading, however, it would be a good idea to refactor our code. Specifically, I would like to extract all localization-related logic to a separate file. Therefore, let's create a new `plugins/Translation.js` file:

```
import { i18n } from '../i18n'

const Trans = {
  get defaultLocale () {
    return process.env.VUE_APP_I18N_LOCALE
  },
  get supportedLocales() {
    return
process.env.VUE_APP_I18N_SUPPORTED_LOCALE.split(',')
  },
  get currentLocale() {
    return i18n.locale
  },
  set currentLocale(locale) {
    i18n.locale = locale
  }
}

export { Trans }
```

There are three getters to read:

- The default locale
- The list of supported locales
- The currently set locale

Also, we have a method to set an attribute to change the current language.

Now, let's implement a method to change the language and load the necessary translation file:

```
const Trans = {
  //  ...
  changeLocale(locale) {
    if (!Trans.isLocaleSupported(locale)) return
Promise.reject(new Error('Locale not supported'))
    if (i18n.locale === locale) return
Promise.resolve(locale)
    return Trans.loadLocaleFile(locale).then(msgs => {
      i18n.setLocaleMessage(locale, msgs.default || msgs)
      return Trans.setI18nLocaleInServices(locale)
    })
  }
}

export { Trans }
```

This method is based on promises. If the locale is not supported, then we just reject with an error message. If the chosen locale is the same, we resolve the promise with that locale. Otherwise, load the corresponding translation file, pass translation messages to Vue I18n, switch the locale, and return it.

Here is the method to check if the requested locale is supported:

```
isLocaleSupported(locale) {
  return Trans.supportedLocales.includes(locale)
}
```

Method to load a translation file:

```
loadLocaleFile(locale) {
 return import(`@/locales/${locale}.json`)
}
```

Method to switch locale:

```
setI18nLocaleInServices(locale) {
  Trans.currentLocale = locale
  document.querySelector('html').setAttribute('lang', locale)
  return locale
}
```

Note that it also updates the `lang` attribute of the `html` tag.

Let's also add a special method that will be used in our routing:

```
routeMiddleware(to, from, next) {
  const locale = to.params.locale
  if (!Trans.isLocaleSupported(locale)) return
next(Trans.defaultLocale)
  return Trans.changeLocale(locale).then(() => next())
}
```

This method has the same logic as the one provided in the `beforeEnter()` method, but it relies on the newly created plugin.

Lastly, I'd like to introduce a special method to generate localized routes. It is going to accept the route name:

```
i18nRoute(to) {
  return {
    ...to,
    params: {locale: this.currentLocale, ...to.params}
  }
}
```

Our plugin is ready so we may refactor the code based on it!

Refactoring I18n Config

The next step is to update our `i18n.js` file to take advantage of the new `Translation` plugin. Here is a new version of this file:

```
import Vue from 'vue'
import VueI18n from 'vue-i18n'
import en from '@/locales/en.json' // <-------------- 1

VueI18n.prototype.getChoiceIndex = function(choice, choicesLength) {
  // ...
}

Vue.use(VueI18n)

const dateTimeFormats = {
  // ...
}

const numberFormats = {
  // ...
}

export const i18n = new VueI18n({ // <-------------- 2
  locale: process.env.VUE_APP_I18N_LOCALE || 'en',
```

```
  fallbackLocale:
process.env.VUE_APP_I18N_FALLBACK_LOCALE || 'en',
  messages: { en }, // <-------------- 3
  dateTimeFormats,
  numberFormats
})
```

Main things to note here:

1. Initially, we are loading only the English translation file (as English is set as the default locale).
2. We are exporting an `i18n` constant instead of using just `export default`.
3. `messages` now contains only the imported English translations.
4. `loadLocaleMessages` function is removed because we have implemented our own loader inside the `Translation` plugin.

Refactoring Main File and Routes

Next, adjust the contents of the `main.js` file in the following way:

```
import Vue from 'vue'
import App from './App.vue'
import { i18n } from './i18n' // <------------- 1
import router from './router'
import { Trans } from './plugins/Translation'

Vue.prototype.$i18nRoute = Trans.i18nRoute.bind(Trans) // <------------- 2

Vue.config.productionTip = false

new Vue({
  i18n,
  router,
  render: h => h(App)
}).$mount('#app')
```

There are two things to note here:

1. Make sure to import all the necessary modules properly.
2. Allow `$i18nRoute` to be used in your templates.

Next, tweak the `router/index.js` file:

```
import Vue from 'vue'
import VueRouter from 'vue-router'
import { Trans } from '@/plugins/Translation' // <-------------- 1

function load(component) {
  // ...
}

Vue.use(VueRouter)

const routes = [{
    path: '/:locale',
    component: {
      template: "<router-view></router-view>"
    },
    beforeEnter: Trans.routeMiddleware, // <---------------- 2
    children: [
      // ...
    ]
  },
  {
    path: '*',
    redirect() {
      return Trans.defaultLocale; // <---------------- 3
    }
  }
]

const router = new VueRouter({
  mode: 'history',
  base: process.env.BASE_URL,
  routes
})

export default router
```

Note the following:

1. We are now importing only the Translation plugin which contains all locale-related logic.
2. `beforeEnter` now delegates to the `routeMiddleware`. This way we are making our code less cluttered and separate the logic into the proper files.
3. The default locale is also fetched using the new plugin.

Great job!

Refactoring Components

We are nearly done with all the refactoring. The last thing to take care of are our components. Start with the `components/LocaleSwitcher.vue`:

```
<template>
<ul>
  <li v-for="locale in supportedLocales" :key="locale" @click="switchLo-
cale(locale)"> <!-- <------------ 1 -->
    {{ $t('menu.' + locale) }} <!-- <------------ 2 -->
  </li>
</ul>
</template>

<script>
import { Trans } from '@/plugins/Translation' // <------------ 3

export default {
  name: 'LocaleSwitcher',
  computed: { <------------ 4
    supportedLocales () {
      return Trans.supportedLocales
    },
  },
  methods: {
    switchLocale(locale) {
      if (this.$i18n.locale !== locale) {
        const to = this.$router.resolve({ params: {locale} })
```

```
        return Trans.changeLocale(locale).then(() => { <------------ 5
          this.$router.push(to.location)
        })
      }
    }
  }
}
</script>
```

Main things to note:

1. We are using the `supportedLocales` attribute which is defined below.
2. We also translate the locale codes.
3. Don't forget to load the `Translation` plugin.
4. Introduce a new `supportedLocales` computed attribute which relies on the `Translation.supportedLocales` getter.
5. Locale is now also changed using our plugin.

New English translations:

```
{
  "menu": {
    "en": "English",
    "ru": "Russian"
    // ...
  }
  // ...
}
```

Russian translations:

```
{
  "menu": {
    "en": "Английский",
    "ru": "Русский"
```

```
    // ...
  }
  // ...
}
```

Finally, adjust the `components/MainMenu.vue`:

```
<template>
<div id="nav">
  <router-link :to="$i18nRoute({ name: 'Home' })">
    {{ $t('menu.home') }}
  </router-link>

  <br>

  <router-link :to="$i18nRoute({ name: 'About' })">
    {{ $t('menu.about') }}
  </router-link>
</div>
</template>
```

Here we are taking advantage of the `$i18nRoute()` helper to generate localized routes based on the currently set locale.

At this point you may boot your application and make sure everything is working!

Reading User Preferred Language

The very last thing I wanted to should you today is the ability to adjust the initial locale based on the user's preferred language. Usually, browsers send preferred language in the request so we can take advantage of this fact. To get started, install axios library which will be used to work with the headers:

```
npm install axios
```

Import axios inside `plugins/Translation.js` file:

```
import axios from 'axios'

// ...
```

Add a new method that will try to read the preferred locale and check whether it is supported:

```
getUserSupportedLocale() {
  const userPreferredLocale = Trans.getUserLocale() // <------------- 1

  if (Trans.isLocaleSupported(userPreferredLocale.locale)) { // <---------
---- 2
    return userPreferredLocale.locale
  }

  if
(Trans.isLocaleSupported(userPreferredLocale.localeNoISO)) { // <---------
---- 3
    return userPreferredLocale.localeNoISO
```

```
  }
  return Trans.defaultlocale // <------------- 4
}
```

The logic of this method is rather simple:

1. We read the preferred locale.
2. If the app supports this locale, return it.
3. If the locale is unknown, try to strip the ISO code and check again.
4. Finally, if both checks were unsuccessful, return the default locale.

Here is the implementation of the getUserLocale() method:

```
getUserLocale() {
  const locale = window.navigator.language || window.navigator.userLan-
guage || Trans.defaultLocale // <----------- 1
  return {
    locale: locale, <-------------- 2
    localeNoISO: locale.split('-')[0] <------------- 3
  }
}
```

1. Try to read the preferred language. If the preferred language is not set, simply use the default locale.
2. Return full locale (for example, en-us) as the first attribute of the object.
3. Return locale without the ISO part (for example, en) as the second attribute.

Having these two methods in place, adjust the routeMiddleware method:

```
routeMiddleware(to, from, next) {
  const locale = to.params.locale
  if (!Trans.isLocaleSupported(locale)) return
next(Trans.getUserSupportedLocale()) // <----------- 1
  return Trans.changeLocale(locale).then(() => next())
}
```

There is only one change. If the requested locale is not supported, we fetch the preferred locale.

Lastly, set the Accept-Language header using axios:

```
setI18nLocaleInServices(locale) {
  Trans.currentLocale = locale
  axios.defaults.headers.common['Accept-Language'] =
locale // <---------------
  document.querySelector('html').setAttribute('lang',
locale)
  return locale
}
```

That's all! Now the application will try to use the language preferred by the current visitor.

Conclusion

In this tutorial we have seen how to implement internationalization and localization using the Vue I18n solution. We have learned how to configure this library, perform translations, use pluralization, and localize datetime and numbers. On top of that, we've added a language switcher, enabled Vue Router, introduced lazy loading, and support for the user preferred locale. Not bad for a single article!

Of course, this demo app can be enhanced further. To get some inspiration you may check the example i18n-starter repo created by Dobromir Hristov. Also, you may be interested in a vue-i18n-extract tool which scans your project and extracts translations into JSON files.

Don't hesitate to contact us if you have any additional questions or concerns.


lokalise