Vue-Router

In a JavaScript web application, a router is the part that syncs the currently displayed view with the browser address bar content.

In other words, it's the part that makes the URL change when you click something in the page, and helps to show the correct view when you hit a specific URL.

Traditionally, the Web is built around URLs. When you hit a certain URL, a specific page is displayed.

With the introduction of applications that run inside the browser and change what the user sees, many applications broke this interaction, and you had to manually update the URL with the browser's History API.

You need a router when you need to sync URLs to views in your app. It's a very common need, and all the major modern frameworks now allow you to manage routing.

The Vue Router library is the way to go for Vue.js applications. Vue does not enforce the use of this library. You can use whatever generic routing library you want, or also create your own History API integration, but the benefit of using Vue Router is that it's official.

This means it's maintained by the same people who maintain Vue, so you get a more consistent integration in the framework, and the guarantee that it's always going to be compatible in the future, no matter what.

Installation

Vue Router is available via npm with the package named vue-router.

If you use Vue via a script tag, you can include Vue Router using

```
<script src="https://unpkg.com/vue-router"></script>
```

If you use the Vue CLI, install it using:

```
npm install vue-router
```

Once you install vue-router and make it available either using a script tag or via Vue CLI, you can now import it in your app.

You import it after vue, and you call Vue.use(VueRouter) to **install** it inside the app:

```
import Vue from 'vue'


import VueRouter from 'vue-router'
Vue.use(VueRouter)
```

After you call Vue.use() passing the router object, in any component of the app you have access to these objects:

- this.$router is the router object
- this.$route is the current route object

The router object

The router object, accessed using **this.$router** from any component when the Vue Router is installed in the root Vue component, offers many nice features.

We can make the app navigate to a new route using

- this.$router.push()
- this.$router.replace()
- this.$router.go()

which resemble the pushState, replaceState and go methods of the History API.

- push() is used to go to a new route, adding a new item to the browser history
- replace() is the same, except it does not push a new state to the history
- go() goes back and forth, accepting a number that can be positive or negative to go back in the history

Defining the routes

A router-link component renders an a tag by default (you can change that). Every time the route changes, either by clicking a link or by changing the URL, a router-link-active class is added to the element that refers to the active route, allowing you to style it.

The router-view component is where the Vue Router will put the content that matches the current URL.

When using the Vue Router, you don’t pass a render property but instead, you use router.

```
new Vue({

 router

}).$mount('#app')
```

is shorthand for:

```
new Vue({

 router: router

}).$mount('#app')
```

What happens when a user clicks a router-link?

The application will render the route component that matches the URL passed to the link.

The new route component that handles the URL is instantiated and its guards called, and the old route component will be destroyed.

Dynamic routing

A very common need is to handle dynamic routes, like having all posts under /post/, each with the slug name:

- /post/first
- /post/another-post
- /post/hello-world

You can achieve this using a dynamic segment.

Those were static segments:

```
const router = new VueRouter({

        routes: [

          { path: '/', component: Home },

          { path: '/login', component: Login },

          { path: '/about', component: About }

        ]

})
```

We add in a dynamic segment to handle blog posts:

```
const router = new VueRouter({

        routes: [

          { path: '/', component: Home },

          { path: '/post/:post_slug', component: Post },

          { path: '/login', component: Login },

          { path: '/about', component: About }

        ]

})
```

Notice the :post_slug syntax. This means that you can use any string, and that will be mapped to the post_slugplaceholder.

You’re not limited to this kind of syntax. Vue relies on this library to parse dynamic routes, and you can go wild with Regular Expressions.

Now inside the Post route component we can reference the route using $route, and the post slug using $route.params.post_slug:

```
const Post = { template: '<div>Post: {{ $route.params.post_slug }}</div>'}
```

You can have as many dynamic segments as you want, in the same URL:

```
/post/:author/:post_slug
```

In the case of dynamic routes, what happens is a little different.

For Vue to be more efficient, instead of destroying the current route component and re-instantiating it, it reuses the current instance.

When this happens, Vue calls the beforeRouteUpdate life cycle event.

Using props

In the examples, I used $route.params.* to access the route data. A component should not be so tightly coupled with the router, and instead, we can use props:

```
const Post = {

 props: ['post_slug'],

 template: '<div>Post: {{ post_slug }}</div>'

}
const router = new VueRouter({
 routes: [
  { path: '/post/:post_slug', component: Post, props: true }
 ]
})
```

Notice the props: true passed to the route object to enable this functionality.

Nested routes

Before I mentioned that you can have as many dynamic segments as you want, in the same URL, like:

```
/post/:author/:post_slug
```

So, say we have an Author component taking care of the first dynamic segment:

```
<template>


 <div id="app">


   <router-view></router-view>


 </div>


</template>
<script>
import Vue from 'vue'
import VueRouter from 'vue-router'
Vue.use(Router)
const Author  = {
 template: '<div>Author: {{ $route.params.author}}</div>'
}
const router = new VueRouter({
 routes: [
   { path: '/post/:author', component: Author }
 ]
})
new Vue({
 router
}).$mount('#app')
</script>
```

We can insert a second router-view component instance inside the Author template:

```
const Author  = {

 template: '<div>Author: {{
$route.params.author}}<router-view></router-view></div>'

}
```

We add the Post component:

```
const Post = {

 template: '<div>Post: {{ $route.params.post_slug }}</div>'

}
```

Then we’ll inject the inner dynamic route in the VueRouter configuration:

```
const router = new VueRouter({

 routes: [{

   path: '/post/:author',

   component: Author,
  children: [

    path: ':post_slug',

    component: Post

  ]

 }]

})
```

Redirections

Redirecting is also done in the routes configuration. To redirect from /a to /b:

```
const router = new VueRouter({
  routes: [
    { path: '/a', redirect: '/b' }
  ]
})
```

The redirect can also be targeting a named route:

```
const router = new VueRouter({
  routes: [
    { path: '/a', redirect: { name: 'foo' }}
  ]
})
```

Alias

A redirect means when the user visits /a, the URL will be replaced by /b, and then matched as /b. But what is an alias?

An alias of /a as /b means when the user visits /b, the URL remains /b, but it will be matched as if the user is visiting /a.

The above can be expressed in the route configuration as:

```
const router = new VueRouter({
  routes: [
    { path: '/a', component: A, alias: '/b' }
  ]
})
```

An alias gives you the freedom to map a UI structure to an arbitrary URL, instead of being constrained by the configuration's nesting structure.

Advanced Topics

Named Routes

Sometimes it is more convenient to identify a route with a name, especially when linking to a route or performing navigations. You can give a route a name in the routes options while creating the Router instance:

```
const router = new VueRouter({
  routes: [
    {
      path: '/user/:userId',
      name: 'user',
      component: User
    }
  ]
})
```

To link to a named route, you can pass an object to the router-link component's to prop:

```
<router-link :to="{ name: 'user', params: { userId: 123 }}">User</router-link>
```

This is the exact same object used programatically with router.push():

```
router.push({ name: 'user', params: { userId: 123 }})
```

In both cases, the router will navigate to the path /user/123.

Navigation Guards

As the name suggests, the navigation guards provided by vue-router are primarily used to guard navigations either by redirecting it or canceling it. There are a number of ways to hook into the route navigation process: globally, per-route, or in-component.

Remember that params or query changes won't trigger enter/leave navigation guards. You can either watch the $route object to react to those changes, or use the beforeRouteUpdate in-component guard

https://router.vuejs.org/guide/advanced/navigation-guards.html#global-resolve-guards

Reacting to Params Changes

One thing to note when using routes with params is that when the user navigates from /user/foo to /user/bar, the same component instance will be reused. Since both routes render the same component, this is more efficient than destroying the old instance and then creating a new one. However, this also means that the lifecycle hooks of the component will not be called.

To react to params changes in the same component, you can simply watch the $route object:

```
const User = {
 template: '...',
 watch: {
  '$route' (to, from) {
   // react to route changes...
  }
 }
}
```

Or, use the beforeRouteUpdate navigation guard introduced in 2.2:

```
const User = {
 template: '...',
 beforeRouteUpdate (to, from, next) {
  // react to route changes...
  // don't forget to call next()
 }
}
```

Transitions

Since the <router-view> is essentially a dynamic component, we can apply transition effects to it the same way using the <transition> component:

```
<transition>
 <router-view></router-view>
</transition>
```

https://vuejs.org/v2/guide/transitions.html