# 第1章 Introduction

Vue Router is the official router for Vue.js. It deeply integrates with Vue.js core to make building Single Page Applications with Vue.js a breeze. Features include:

Nested routes mapping

Dynamic Routing

Modular, component-based router configuration

Route params, query, wildcards

View transition effects powered by Vue.js' transition system

Fine-grained navigation control

Links with automatic active CSS classes

HTML5 history mode or hash mode

Customizable Scroll Behavior

Proper encoding for URLs

# 第2章 Installation

Direct Download / CDN#

https://unpkg.com/vue-router@4

Unpkg.com provides npm-based CDN links. The above link will always point to the latest release on npm. You can also use a specific version/tag via URLs like https://unpkg.com/vue-router@4.0.15/dist/vue-router.global.js.

npm#

npm install vue-router@4

yarn#

yarn add vue-router@4

Suggest changes to this page

# 第3章 Getting Started

Creating a Single-page Application with Vue + Vue Router feels natural: with Vue.js, we are already composing our application with components. When adding Vue Router to the mix, all we need to do is map our components to the routes and let Vue Router know where to render them. Here's a basic example:

HTML#

<script src="https://unpkg.com/vue@3"></script>

<script src="https://unpkg.com/vue-router@4"></script>

<div id="app">

<h1>Hello App!</h1>

<p>

<!-- use the router-link component for navigation. -->

<!-- specify the link by passing the `to` prop. -->

<!-- `<router-link>` will render an `<a>` tag with the correct `href` attribute -->

<router-link to="/">Go to Home</router-link>

<router-link to="/about">Go to About</router-link>

</p>

<!-- route outlet -->

<!-- component matched by the route will render here -->

<router-view></router-view>

</div>

router-link#

Note how instead of using regular a tags, we use a custom component router-link to create links. This allows Vue Router to change the URL without reloading the page, handle URL generation as well as its encoding. We will see later how to benefit from these features.

router-view#

router-view will display the component that corresponds to the url. You can put it anywhere to adapt it to your layout.

JavaScript#

// 1. Define route components.

// These can be imported from other files

const Home = { template: '<div>Home</div>' }

const About = { template: '<div>About</div>' }

// 2. Define some routes

// Each route should map to a component.

// We'll talk about nested routes later.

const routes = [

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

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

]

// 3. Create the router instance and pass the `routes` option

// You can pass in additional options here, but let's

// keep it simple for now.

const router = VueRouter.createRouter({

// 4. Provide the history implementation to use. We are using the hash history for simplicity here.

history: VueRouter.createWebHashHistory(),

routes, // short for `routes: routes`

})

// 5. Create and mount the root instance.

const app = Vue.createApp({})

// Make sure to \_use\_ the router instance to make the

// whole app router-aware.

app.use(router)

app.mount('#app')

// Now the app has started!

By calling app.use(router), we get access to it as this.$router as well as the current route as this.$route inside of any component:

// Home.vue

export default {

computed: {

username() {

// We will see what `params` is shortly

return this.$route.params.username

},

},

methods: {

goToDashboard() {

if (isAuthenticated) {

this.$router.push('/dashboard')

} else {

this.$router.push('/login')

}

},

},

}

To access the router or the route inside the setup function, call the useRouter or useRoute functions. We will learn more about this in the Composition API

Throughout the docs, we will often use the router instance. Keep in mind that this.$router is exactly the same as directly using the router instance created through createRouter. The reason we use this.$router is because we don't want to import the router in every single component that needs to manipulate routing.

# 第4章 Dynamic Route Matching with Params

Very often we will need to map routes with the given pattern to the same component. For example we may have a User component which should be rendered for all users but with different user IDs. In Vue Router we can use a dynamic segment in the path to achieve that, we call that a param:

const User = {

template: '<div>User</div>',

}

// these are passed to `createRouter`

const routes = [

// dynamic segments start with a colon

{ path: '/users/:id', component: User },

]

Now URLs like /users/johnny and /users/jolyne will both map to the same route.

A param is denoted by a colon :. When a route is matched, the value of its params will be exposed as this.$route.params in every component. Therefore, we can render the current user ID by updating User's template to this:

const User = {

template: '<div>User {{ $route.params.id }}</div>',

}

You can have multiple params in the same route, and they will map to corresponding fields on $route.params. Examples:

pattern matched path $route.params

/users/:username /users/eduardo { username: 'eduardo' }

/users/:username/posts/:postId /users/eduardo/posts/123 { username: 'eduardo', postId: '123' }

In addition to $route.params, the $route object also exposes other useful information such as $route.query (if there is a query in the URL), $route.hash, etc. You can check out the full details in the API Reference.

A working demo of this example can be found here.

Reacting to Params Changes#

Watch a free video lesson on Vue School

One thing to note when using routes with params is that when the user navigates from /users/johnny to /users/jolyne, 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 anything on the $route object, in this scenario, the $route.params:

const User = {

template: '...',

created() {

this.$watch(

() => this.$route.params,

(toParams, previousParams) => {

// react to route changes...

}

)

},

}

Or, use the beforeRouteUpdate navigation guard, which also allows to cancel the navigation:

const User = {

template: '...',

async beforeRouteUpdate(to, from) {

// react to route changes...

this.userData = await fetchUser(to.params.id)

},

}

Catch all / 404 Not found Route#

Watch a free video lesson on Vue School

Regular params will only match characters in between url fragments, separated by /. If we want to match anything, we can use a custom param regexp by adding the regexp inside parentheses right after the param:

const routes = [

// will match everything and put it under `$route.params.pathMatch`

{ path: '/:pathMatch(.\*)\*', name: 'NotFound', component: NotFound },

// will match anything starting with `/user-` and put it under `$route.params.afterUser`

{ path: '/user-:afterUser(.\*)', component: UserGeneric },

]

In this specific scenario we are using a custom regexp between parentheses and marking the pathMatch param as optionally repeatable. This allows us to directly navigate to the route if we need to by splitting the path into an array:

this.$router.push({

name: 'NotFound',

// preserve current path and remove the first char to avoid the target URL starting with `//`

params: { pathMatch: this.$route.path.substring(1).split('/') },

// preserve existing query and hash if any

query: this.$route.query,

hash: this.$route.hash,

})

See more in the repeated params section.

If you are using History mode, make sure to follow the instructions to correctly configure your server as well.

Advanced Matching Patterns#

Vue Router uses its own path matching syntax, inspired by the one used by express, so it supports many advanced matching patterns such as optional params, zero or more / one or more requirements, and even custom regex patterns. Please check the Advanced Matching documentation to explore them.

# 第5章 Routes' Matching Syntax

Most applications will use static routes like /about and dynamic routes like /users/:userId like we just saw in Dynamic Route Matching, but Vue Router has much more to offer!

TIP

For the sake of simplicity, all route records are omitting the component property to focus on the path value.

Custom regex in params#

When defining a param like :userId, we internally use the following regex ([^/]+) (at least one character that isn't a slash /) to extract params from URLs. This works well unless you need to differentiate two routes based on the param content. Imagine two routes /:orderId and /:productName, both would match the exact same URLs, so we need a way to differentiate them. The easiest way would be to add a static section to the path that differentiates them:

const routes = [

// matches /o/3549

{ path: '/o/:orderId' },

// matches /p/books

{ path: '/p/:productName' },

]

But in some scenarios we don't want to add that static section /o/p. However, orderId is always a number while productName can be anything, so we can specify a custom regex for a param in parentheses:

const routes = [

// /:orderId -> matches only numbers

{ path: '/:orderId(\\d+)' },

// /:productName -> matches anything else

{ path: '/:productName' },

]

Now, going to /25 will match /:orderId while going to anything else will match /:productName. The order of the routes array doesn't even matter!

TIP

Make sure to escape backslashes (\) like we did with \d (becomes \\d) to actually pass the backslash character in a string in JavaScript.

Repeatable params#

If you need to match routes with multiple sections like /first/second/third, you should mark a param as repeatable with \* (0 or more) and + (1 or more):

const routes = [

// /:chapters -> matches /one, /one/two, /one/two/three, etc

{ path: '/:chapters+' },

// /:chapters -> matches /, /one, /one/two, /one/two/three, etc

{ path: '/:chapters\*' },

]

This will give you an array of params instead of a string and will also require you to pass an array when using named routes:

// given { path: '/:chapters\*', name: 'chapters' },

router.resolve({ name: 'chapters', params: { chapters: [] } }).href

// produces /

router.resolve({ name: 'chapters', params: { chapters: ['a', 'b'] } }).href

// produces /a/b

// given { path: '/:chapters+', name: 'chapters' },

router.resolve({ name: 'chapters', params: { chapters: [] } }).href

// throws an Error because `chapters` is empty

These can also be combined with a custom regex by adding them after the closing parentheses:

const routes = [

// only match numbers

// matches /1, /1/2, etc

{ path: '/:chapters(\\d+)+' },

// matches /, /1, /1/2, etc

{ path: '/:chapters(\\d+)\*' },

]

Sensitive and strict route options#

By default, all routes are case-insensitive and match routes with or without a trailing slash. e.g. a route /users matches /users, /users/, and even /Users/. This behavior can be configured with the strict and sensitive options, they can be set both at a router and route level:

const router = createRouter({

history: createWebHistory(),

routes: [

// will match /users/posva but not:

// - /users/posva/ because of strict: true

// - /Users/posva because of sensitive: true

{ path: '/users/:id', sensitive: true },

// will match /users, /Users, and /users/42 but not /users/ or /users/42/

{ path: '/users/:id?' },

],

strict: true, // applies to all routes

})

Optional parameters#

You can also mark a parameter as optional by using the ? modifier (0 or 1):

const routes = [

// will match /users and /users/posva

{ path: '/users/:userId?' },

// will match /users and /users/42

{ path: '/users/:userId(\\d+)?' },

]

Note that \* technically also marks a parameter as optional but ? parameters cannot be repeated.

Debugging#

If you need to dig how your routes are transformed into a regex to understand why a route isn't being matched or, to report a bug, you can use the path ranker tool. It supports sharing your routes through the URL.

# 第6章 Nested Routes

Some application's UIs are composed of components that are nested multiple levels deep. In this case, it is very common that the segments of a URL corresponds to a certain structure of nested components, for example:

/user/johnny/profile /user/johnny/posts

+------------------+ +-----------------+

| User | | User |

| +--------------+ | | +-------------+ |

| | Profile | | +------------> | | Posts | |

| | | | | | | |

| +--------------+ | | +-------------+ |

+------------------+ +-----------------+

With Vue Router, you can express this relationship using nested route configurations.

Given the app we created in the last chapter:

<div id="app">

<router-view></router-view>

</div>

const User = {

template: '<div>User {{ $route.params.id }}</div>',

}

// these are passed to `createRouter`

const routes = [{ path: '/user/:id', component: User }]

The <router-view> here is a top-level router-view. It renders the component matched by a top level route. Similarly, a rendered component can also contain its own, nested <router-view>. For example, if we add one inside the User component's template:

const User = {

template: `

<div class="user">

<h2>User {{ $route.params.id }}</h2>

<router-view></router-view>

</div>

`,

}

To render components into this nested router-view, we need to use the children option in any of the routes:

const routes = [

{

path: '/user/:id',

component: User,

children: [

{

// UserProfile will be rendered inside User's <router-view>

// when /user/:id/profile is matched

path: 'profile',

component: UserProfile,

},

{

// UserPosts will be rendered inside User's <router-view>

// when /user/:id/posts is matched

path: 'posts',

component: UserPosts,

},

],

},

]

Note that nested paths that start with / will be treated as a root path. This allows you to leverage the component nesting without having to use a nested URL.

As you can see the children option is just another Array of routes like routes itself. Therefore, you can keep nesting views as much as you need.

At this point, with the above configuration, when you visit /user/eduardo, nothing will be rendered inside User's router-view, because no nested route is matched. Maybe you do want to render something there. In such case you can provide an empty nested path:

const routes = [

{

path: '/user/:id',

component: User,

children: [

// UserHome will be rendered inside User's <router-view>

// when /user/:id is matched

{ path: '', component: UserHome },

// ...other sub routes

],

},

]

A working demo of this example can be found here.

Nested Named Routes#

When dealing with Named Routes, you usually name the children routes:

const routes = [

{

path: '/user/:id',

component: User,

// notice how only the child route has a name

children: [{ path: '', name: 'user', component: UserHome }],

},

]

This will ensure navigating to /user/:id will always display the nested route.

In some scenarios, you may want to navigate to a named route without navigating to the nested route. For example, if you want to navigate to /user/:id without displaying the nested route. In that case, you can also name the parent route but note that reloading the page will always display the nested child as it's considered a navigation to the path /users/:id instead of the named route:

const routes = [

{

path: '/user/:id',

name: 'user-parent'

component: User,

children: [{ path: '', name: 'user', component: UserHome }],

},

]

# 第7章 Programmatic Navigation

Aside from using <router-link> to create anchor tags for declarative navigation, we can do this programmatically using the router's instance methods.

Navigate to a different location#

Note: Inside of a Vue instance, you have access to the router instance as $router. You can therefore call this.$router.push.

To navigate to a different URL, use router.push. This method pushes a new entry into the history stack, so when the user clicks the browser back button they will be taken to the previous URL.

This is the method called internally when you click a <router-link>, so clicking <router-link :to="..."> is the equivalent of calling router.push(...).

Declarative Programmatic

<router-link :to="..."> router.push(...)

The argument can be a string path, or a location descriptor object. Examples:

// literal string path

router.push('/users/eduardo')

// object with path

router.push({ path: '/users/eduardo' })

// named route with params to let the router build the url

router.push({ name: 'user', params: { username: 'eduardo' } })

// with query, resulting in /register?plan=private

router.push({ path: '/register', query: { plan: 'private' } })

// with hash, resulting in /about#team

router.push({ path: '/about', hash: '#team' })

Note: params are ignored if a path is provided, which is not the case for query, as shown in the example above. Instead, you need to provide the name of the route or manually specify the whole path with any parameter:

const username = 'eduardo'

// we can manually build the url but we will have to handle the encoding ourselves

router.push(`/user/${username}`) // -> /user/eduardo

// same as

router.push({ path: `/user/${username}` }) // -> /user/eduardo

// if possible use `name` and `params` to benefit from automatic URL encoding

router.push({ name: 'user', params: { username } }) // -> /user/eduardo

// `params` cannot be used alongside `path`

router.push({ path: '/user', params: { username } }) // -> /user

When specifying params, make sure to either provide a string or number (or an array of these for repeatable params). Any other type (like undefined, false, etc) will be automatically stringified. For optional params, you can provide an empty string ("") as the value to skip it.

Since the prop to accepts the same kind of object as router.push, the exact same rules apply to both of them.

router.push and all the other navigation methods return a Promise that allows us to wait till the navigation is finished and to know if it succeeded or failed. We will talk more about that in Navigation Handling.

Replace current location#

It acts like router.push, the only difference is that it navigates without pushing a new history entry, as its name suggests - it replaces the current entry.

Declarative Programmatic

<router-link :to="..." replace> router.replace(...)

It's also possible to directly add a property replace: true to the routeLocation that is passed to router.push:

router.push({ path: '/home', replace: true })

// equivalent to

router.replace({ path: '/home' })

Traverse history#

Watch a free video lesson on Vue School

This method takes a single integer as parameter that indicates by how many steps to go forward or go backward in the history stack, similar to window.history.go(n).

Examples

// go forward by one record, the same as router.forward()

router.go(1)

// go back by one record, the same as router.back()

router.go(-1)

// go forward by 3 records

router.go(3)

// fails silently if there aren't that many records

router.go(-100)

router.go(100)

History Manipulation#

You may have noticed that router.push, router.replace and router.go are counterparts of window.history.pushState, window.history.replaceState and window.history.go, and they do imitate the window.history APIs.

Therefore, if you are already familiar with Browser History APIs, manipulating history will feel familiar when using Vue Router.

It is worth mentioning that Vue Router navigation methods (push, replace, go) work consistently no matter the kind of history option is passed when creating the router instance.

# 第8章 Named Routes

Alongside the path, you can provide a name to any route. This has the following advantages:

No hardcoded URLs

Automatic encoding/decoding of params

Prevents you from having a typo in the url

Bypassing path ranking (e.g. to display a )

const routes = [

{

path: '/user/:username',

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: { username: 'erina' }}">

User

</router-link>

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

router.push({ name: 'user', params: { username: 'erina' } })

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

Full example here.

# 第9章 Named Views

Sometimes you need to display multiple views at the same time instead of nesting them, e.g. creating a layout with a sidebar view and a main view. This is where named views come in handy. Instead of having one single outlet in your view, you can have multiple and give each of them a name. A router-view without a name will be given default as its name.

<router-view class="view left-sidebar" name="LeftSidebar"></router-view>

<router-view class="view main-content"></router-view>

<router-view class="view right-sidebar" name="RightSidebar"></router-view>

A view is rendered by using a component, therefore multiple views require multiple components for the same route. Make sure to use the components (with an s) option:

const router = createRouter({

history: createWebHashHistory(),

routes: [

{

path: '/',

components: {

default: Home,

// short for LeftSidebar: LeftSidebar

LeftSidebar,

// they match the `name` attribute on `<router-view>`

RightSidebar,

},

},

],

})

A working demo of this example can be found here.

Nested Named Views#

It is possible to create complex layouts using named views with nested views. When doing so, you will also need to give nested router-view a name. Let's take a Settings panel example:

/settings/emails /settings/profile

+-----------------------------------+ +------------------------------+

| UserSettings | | UserSettings |

| +-----+-------------------------+ | | +-----+--------------------+ |

| | Nav | UserEmailsSubscriptions | | +------------> | | Nav | UserProfile | |

| | +-------------------------+ | | | +--------------------+ |

| | | | | | | | UserProfilePreview | |

| +-----+-------------------------+ | | +-----+--------------------+ |

+-----------------------------------+ +------------------------------+

Nav is just a regular component

UserSettings is the parent view component

UserEmailsSubscriptions, UserProfile, UserProfilePreview are nested view components

Note: Let's forget about how the HTML/CSS should look like to represent such layout and focus on the components used.

The <template> section for UserSettings component in the above layout would look something like this:

<!-- UserSettings.vue -->

<div>

<h1>User Settings</h1>

<NavBar />

<router-view />

<router-view name="helper" />

</div>

Then you can achieve the layout above with this route configuration:

{

path: '/settings',

// You could also have named views at the top

component: UserSettings,

children: [{

path: 'emails',

component: UserEmailsSubscriptions

}, {

path: 'profile',

components: {

default: UserProfile,

helper: UserProfilePreview

}

}]

}

A working demo of this example can be found here.

# 第10章 Redirect and Alias

Redirect#

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

const routes = [{ path: '/home', redirect: '/' }]

The redirect can also be targeting a named route:

const routes = [{ path: '/home', redirect: { name: 'homepage' } }]

Or even use a function for dynamic redirecting:

const routes = [

{

// /search/screens -> /search?q=screens

path: '/search/:searchText',

redirect: to => {

// the function receives the target route as the argument

// we return a redirect path/location here.

return { path: '/search', query: { q: to.params.searchText } }

},

},

{

path: '/search',

// ...

},

]

Note that Navigation Guards are not applied on the route that redirects, only on its target. e.g. In the above example, adding a beforeEnter guard to the /home route would not have any effect.

When writing a redirect, you can omit the component option because it is never directly reached so there is no component to render. The only exception are nested routes: if a route record has children and a redirect property, it should also have a component property.

Relative redirecting#

It's also possible to redirect to a relative location:

const routes = [

{

// will always redirect /users/123/posts to /users/123/profile

path: '/users/:id/posts',

redirect: to => {

// the function receives the target route as the argument

// a relative location doesn't start with `/`

// or { path: 'profile'}

return 'profile'

},

},

]

Alias#

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

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

The above can be expressed in the route configuration as:

const routes = [{ path: '/', component: Homepage, alias: '/home' }]

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. Make the alias start with a / to make the path absolute in nested routes. You can even combine both and provide multiple aliases with an array:

const routes = [

{

path: '/users',

component: UsersLayout,

children: [

// this will render the UserList for these 3 URLs

// - /users

// - /users/list

// - /people

{ path: '', component: UserList, alias: ['/people', 'list'] },

],

},

]

If your route has parameters, make sure to include them in any absolute alias:

const routes = [

{

path: '/users/:id',

component: UsersByIdLayout,

children: [

// this will render the UserDetails for these 3 URLs

// - /users/24

// - /users/24/profile

// - /24

{ path: 'profile', component: UserDetails, alias: ['/:id', ''] },

],

},

]

Note about SEO: when using aliases, make sure to define canonical links.

# 第11章 Passing Props to Route Components

Using $route in your component creates a tight coupling with the route which limits the flexibility of the component as it can only be used on certain URLs. While this is not necessarily a bad thing, we can decouple this behavior with a props option:

We can replace

const User = {

template: '<div>User {{ $route.params.id }}</div>'

}

const routes = [{ path: '/user/:id', component: User }]

with

const User = {

// make sure to add a prop named exactly like the route param

props: ['id'],

template: '<div>User {{ id }}</div>'

}

const routes = [{ path: '/user/:id', component: User, props: true }]

This allows you to use the component anywhere, which makes the component easier to reuse and test.

Boolean mode#

When props is set to true, the route.params will be set as the component props.

Named views#

For routes with named views, you have to define the props option for each named view:

const routes = [

{

path: '/user/:id',

components: { default: User, sidebar: Sidebar },

props: { default: true, sidebar: false }

}

]

Object mode#

When props is an object, this will be set as the component props as-is. Useful for when the props are static.

const routes = [

{

path: '/promotion/from-newsletter',

component: Promotion,

props: { newsletterPopup: false }

}

]

Function mode#

You can create a function that returns props. This allows you to cast parameters into other types, combine static values with route-based values, etc.

const routes = [

{

path: '/search',

component: SearchUser,

props: route => ({ query: route.query.q })

}

]

The URL /search?q=vue would pass {query: 'vue'} as props to the SearchUser component.

Try to keep the props function stateless, as it's only evaluated on route changes. Use a wrapper component if you need state to define the props, that way vue can react to state changes.

# 第12章 Different History modes

The history option when creating the router instance allows us to choose among different history modes.

Hash Mode#

The hash history mode is created with createWebHashHistory():

import { createRouter, createWebHashHistory } from 'vue-router'

const router = createRouter({

history: createWebHashHistory(),

routes: [

//...

],

})

It uses a hash character (#) before the actual URL that is internally passed. Because this section of the URL is never sent to the server, it doesn't require any special treatment on the server level. It does however have a bad impact in SEO. If that's a concern for you, use the HTML5 history mode.

HTML5 Mode#

The HTML5 mode is created with createWebHistory() and is the recommended mode:

import { createRouter, createWebHistory } from 'vue-router'

const router = createRouter({

history: createWebHistory(),

routes: [

//...

],

})

When using createWebHistory(), the URL will look "normal," e.g. https://example.com/user/id. Beautiful!

Here comes a problem, though: Since our app is a single page client side app, without a proper server configuration, the users will get a 404 error if they access https://example.com/user/id directly in their browser. Now that's ugly.

Not to worry: To fix the issue, all you need to do is add a simple catch-all fallback route to your server. If the URL doesn't match any static assets, it should serve the same index.html page that your app lives in. Beautiful, again!

Memory mode#

The memory history mode doesn't assume a browser environment and therefore doesn't interact with the URL nor automatically triggers the initial navigation. This makes it perfect for Node environment and SSR. It is created with createMemoryHistory() and requires you to push the initial navigation after calling app.use(router).

import { createRouter, createMemoryHistory } from 'vue-router'

const router = createRouter({

history: createMemoryHistory(),

routes: [

//...

],

})

While it's not recommended, you can use this mode inside Browser applications but note there will be no history, meaning you won't be able to go back or forward.

Example Server Configurations#

Note: The following examples assume you are serving your app from the root folder. If you deploy to a subfolder, you should use the publicPath option of Vue CLI and the related base property of the router. You also need to adjust the examples below to use the subfolder instead of the root folder (e.g. replacing RewriteBase / with RewriteBase /name-of-your-subfolder/).

Apache#

<IfModule mod\_negotiation.c>

Options -MultiViews

</IfModule>

<IfModule mod\_rewrite.c>

RewriteEngine On

RewriteBase /

RewriteRule ^index\.html$ - [L]

RewriteCond %{REQUEST\_FILENAME} !-f

RewriteCond %{REQUEST\_FILENAME} !-d

RewriteRule . /index.html [L]

</IfModule>

Instead of mod\_rewrite, you could also use FallbackResource.

nginx#

location / {

try\_files $uri $uri/ /index.html;

}

Native Node.js#

const http = require('http')

const fs = require('fs')

const httpPort = 80

http

.createServer((req, res) => {

fs.readFile('index.html', 'utf-8', (err, content) => {

if (err) {

console.log('We cannot open "index.html" file.')

}

res.writeHead(200, {

'Content-Type': 'text/html; charset=utf-8',

})

res.end(content)

})

})

.listen(httpPort, () => {

console.log('Server listening on: http://localhost:%s', httpPort)

})

Express with Node.js#

For Node.js/Express, consider using connect-history-api-fallback middleware.

Internet Information Services (IIS)#

Install IIS UrlRewrite

Create a web.config file in the root directory of your site with the following:

<?xml version="1.0" encoding="UTF-8"?>

<configuration>

<system.webServer>

<rewrite>

<rules>

<rule name="Handle History Mode and custom 404/500" stopProcessing="true">

<match url="(.\*)" />

<conditions logicalGrouping="MatchAll">

<add input="{REQUEST\_FILENAME}" matchType="IsFile" negate="true" />

<add input="{REQUEST\_FILENAME}" matchType="IsDirectory" negate="true" />

</conditions>

<action type="Rewrite" url="/" />

</rule>

</rules>

</rewrite>

</system.webServer>

</configuration>

Caddy v2#

try\_files {path} /

Caddy v1#

rewrite {

regexp .\*

to {path} /

}

Firebase hosting#

Add this to your firebase.json:

{

"hosting": {

"public": "dist",

"rewrites": [

{

"source": "\*\*",

"destination": "/index.html"

}

]

}

}

Netlify#

Create a \_redirects file that is included with your deployed files:

/\* /index.html 200

In vue-cli, nuxt, and vite projects, this file usually goes under a folder named static or public.

You can more about the syntax on Netlify documentation. You can also create a netlify.toml to combine redirections with other Netlify features.

Vercel#

Create a vercel.json file under the root directory of your project with the following:

{

"rewrites": [{ "source": "/:path\*", "destination": "/index.html" }]

}

Caveat#

There is a caveat to this: Your server will no longer report 404 errors as all not-found paths now serve up your index.html file. To get around the issue, you should implement a catch-all route within your Vue app to show a 404 page:

const router = createRouter({

history: createWebHistory(),

routes: [{ path: '/:pathMatch(.\*)', component: NotFoundComponent }],

})

Alternatively, if you are using a Node.js server, you can implement the fallback by using the router on the server side to match the incoming URL and respond with 404 if no route is matched. Check out the Vue server side rendering documentation for more information.

# 第13章 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.

Global Before Guards#

You can register global before guards using router.beforeEach:

const router = createRouter({ ... })

router.beforeEach((to, from) => {

// ...

// explicitly return false to cancel the navigation

return false

})

Global before guards are called in creation order, whenever a navigation is triggered. Guards may be resolved asynchronously, and the navigation is considered pending before all hooks have been resolved.

Every guard function receives two arguments:

to: the target route location in a normalized format being navigated to.

from: the current route location in a normalized format being navigated away from.

And can optionally return any of the following values:

false: cancel the current navigation. If the browser URL was changed (either manually by the user or via back button), it will be reset to that of the from route.

A Route Location: Redirect to a different location by passing a route location as if you were calling router.push(), which allows you to pass options like replace: true or name: 'home'. The current navigation is dropped and a new one is created with the same from.

router.beforeEach(async (to, from) => {

if (

// make sure the user is authenticated

!isAuthenticated &&

// ❗️ Avoid an infinite redirect

to.name !== 'Login'

) {

// redirect the user to the login page

return { name: 'Login' }

}

})

It's also possible to throw an Error if an unexpected situation was met. This will also cancel the navigation and call any callback registered via router.onError().

If nothing, undefined or true is returned, the navigation is validated, and the next navigation guard is called.

All of the the things above work the same way with async functions and Promises:

router.beforeEach(async (to, from) => {

// canUserAccess() returns `true` or `false`

const canAccess = await canUserAccess(to)

if (!canAccess) return '/login'

})

Optional third argument next#

In previous versions of Vue Router, it was also possible to use a third argument next, this was a common source of mistakes and went through an RFC to remove it. However, it is still supported, meaning you can pass a third argument to any navigation guard. In that case, you must call next exactly once in any given pass through a navigation guard. It can appear more than once, but only if the logical paths have no overlap, otherwise the hook will never be resolved or produce errors. Here is a bad example of redirecting to user to /login if they are not authenticated:

// BAD

router.beforeEach((to, from, next) => {

if (to.name !== 'Login' && !isAuthenticated) next({ name: 'Login' })

// if the user is not authenticated, `next` is called twice

next()

})

Here is the correct version:

// GOOD

router.beforeEach((to, from, next) => {

if (to.name !== 'Login' && !isAuthenticated) next({ name: 'Login' })

else next()

})

Global Resolve Guards#

You can register a global guard with router.beforeResolve. This is similar to router.beforeEach because it triggers on every navigation, but resolve guards are called right before the navigation is confirmed, after all in-component guards and async route components are resolved. Here is an example that ensures the user has given access to the Camera for routes that have defined a custom meta property requiresCamera:

router.beforeResolve(async to => {

if (to.meta.requiresCamera) {

try {

await askForCameraPermission()

} catch (error) {

if (error instanceof NotAllowedError) {

// ... handle the error and then cancel the navigation

return false

} else {

// unexpected error, cancel the navigation and pass the error to the global handler

throw error

}

}

}

})

router.beforeResolve is the ideal spot to fetch data or do any other operation that you want to avoid doing if the user cannot enter a page.

Global After Hooks#

You can also register global after hooks, however unlike guards, these hooks do not get a next function and cannot affect the navigation:

router.afterEach((to, from) => {

sendToAnalytics(to.fullPath)

})

They are useful for analytics, changing the title of the page, accessibility features like announcing the page and many other things.

They also reflect navigation failures as the third argument:

router.afterEach((to, from, failure) => {

if (!failure) sendToAnalytics(to.fullPath)

})

Learn more about navigation failures on its guide.

Per-Route Guard#

You can define beforeEnter guards directly on a route's configuration object:

const routes = [

{

path: '/users/:id',

component: UserDetails,

beforeEnter: (to, from) => {

// reject the navigation

return false

},

},

]

beforeEnter guards only trigger when entering the route, they don't trigger when the params, query or hash change e.g. going from /users/2 to /users/3 or going from /users/2#info to /users/2#projects. They are only triggered when navigating from a different route.

You can also pass an array of functions to beforeEnter, this is useful when reusing guards for different routes:

function removeQueryParams(to) {

if (Object.keys(to.query).length)

return { path: to.path, query: {}, hash: to.hash }

}

function removeHash(to) {

if (to.hash) return { path: to.path, query: to.query, hash: '' }

}

const routes = [

{

path: '/users/:id',

component: UserDetails,

beforeEnter: [removeQueryParams, removeHash],

},

{

path: '/about',

component: UserDetails,

beforeEnter: [removeQueryParams],

},

]

Note it is possible to achieve a similar behavior by using route meta fields and global navigation guards.

In-Component Guards#

Finally, you can directly define route navigation guards inside route components (the ones passed to the router configuration)

Using the options API#

You can add the following options to route components:

beforeRouteEnter

beforeRouteUpdate

beforeRouteLeave

const UserDetails = {

template: `...`,

beforeRouteEnter(to, from) {

// called before the route that renders this component is confirmed.

// does NOT have access to `this` component instance,

// because it has not been created yet when this guard is called!

},

beforeRouteUpdate(to, from) {

// called when the route that renders this component has changed, but this component is reused in the new route.

// For example, given a route with params `/users/:id`, when we navigate between `/users/1` and `/users/2`,

// the same `UserDetails` component instance will be reused, and this hook will be called when that happens.

// Because the component is mounted while this happens, the navigation guard has access to `this` component instance.

},

beforeRouteLeave(to, from) {

// called when the route that renders this component is about to be navigated away from.

// As with `beforeRouteUpdate`, it has access to `this` component instance.

},

}

The beforeRouteEnter guard does NOT have access to this, because the guard is called before the navigation is confirmed, thus the new entering component has not even been created yet.

However, you can access the instance by passing a callback to next. The callback will be called when the navigation is confirmed, and the component instance will be passed to the callback as the argument:

beforeRouteEnter (to, from, next) {

next(vm => {

// access to component public instance via `vm`

})

}

Note that beforeRouteEnter is the only guard that supports passing a callback to next. For beforeRouteUpdate and beforeRouteLeave, this is already available, so passing a callback is unnecessary and therefore not supported:

beforeRouteUpdate (to, from) {

// just use `this`

this.name = to.params.name

}

The leave guard is usually used to prevent the user from accidentally leaving the route with unsaved edits. The navigation can be canceled by returning false.

beforeRouteLeave (to, from) {

const answer = window.confirm('Do you really want to leave? you have unsaved changes!')

if (!answer) return false

}

Using the composition API#

If you are writing your component using the composition API and a setup function, you can add update and leave guards through onBeforeRouteUpdate and onBeforeRouteLeave respectively. Please refer to the Composition API section for more details.

The Full Navigation Resolution Flow#

Navigation triggered.

Call beforeRouteLeave guards in deactivated components.

Call global beforeEach guards.

Call beforeRouteUpdate guards in reused components.

Call beforeEnter in route configs.

Resolve async route components.

Call beforeRouteEnter in activated components.

Call global beforeResolve guards.

Navigation is confirmed.

Call global afterEach hooks.

DOM updates triggered.

Call callbacks passed to next in beforeRouteEnter guards with instantiated instances.

# 第15章 Route Meta Fields

Sometimes, you might want to attach arbitrary information to routes like transition names, who can access the route, etc. This can be achieved through the meta property which accepts an object of properties and can be accessed on the route location and navigation guards. You can define meta properties like this:

const routes = [

{

path: '/posts',

component: PostsLayout,

children: [

{

path: 'new',

component: PostsNew,

// only authenticated users can create posts

meta: { requiresAuth: true }

},

{

path: ':id',

component: PostsDetail,

// anybody can read a post

meta: { requiresAuth: false }

}

]

}

]

So how do we access this meta field?

First, each route object in the routes configuration is called a route record. Route records may be nested. Therefore when a route is matched, it can potentially match more than one route record.

For example, with the above route config, the URL /posts/new will match both the parent route record (path: '/posts') and the child route record (path: 'new').

All route records matched by a route are exposed on the $route object (and also route objects in navigation guards) as the $route.matched Array. We could loop through that array to check all meta fields, but Vue Router also provides you a $route.meta that is a non-recursive merge of all meta fields from parent to child. Meaning you can simply write

router.beforeEach((to, from) => {

// instead of having to check every route record with

// to.matched.some(record => record.meta.requiresAuth)

if (to.meta.requiresAuth && !auth.isLoggedIn()) {

// this route requires auth, check if logged in

// if not, redirect to login page.

return {

path: '/login',

// save the location we were at to come back later

query: { redirect: to.fullPath },

}

}

})

TypeScript#

It is possible to type the meta field by extending the RouteMeta interface from vue-router:

// This can be directly added to any of your `.ts` files like `router.ts`

// It can also be added to a `.d.ts` file, in which case you will need to add an export

// to ensure it is treated as a module

export {}

import 'vue-router'

declare module 'vue-router' {

interface RouteMeta {

// is optional

isAdmin?: boolean

// must be declared by every route

requiresAuth: boolean

}

}

# 第16章 Data Fetching

Sometimes you need to fetch data from the server when a route is activated. For example, before rendering a user profile, you need to fetch the user's data from the server. We can achieve this in two different ways:

Fetching After Navigation: perform the navigation first, and fetch data in the incoming component's lifecycle hook. Display a loading state while data is being fetched.

Fetching Before Navigation: Fetch data before navigation in the route enter guard, and perform the navigation after data has been fetched.

Technically, both are valid choices - it ultimately depends on the user experience you are aiming for.

Fetching After Navigation#

When using this approach, we navigate and render the incoming component immediately, and fetch data in the component's created hook. It gives us the opportunity to display a loading state while the data is being fetched over the network, and we can also handle loading differently for each view.

Let's assume we have a Post component that needs to fetch the data for a post based on $route.params.id:

<template>

<div class="post">

<div v-if="loading" class="loading">Loading...</div>

<div v-if="error" class="error">{{ error }}</div>

<div v-if="post" class="content">

<h2>{{ post.title }}</h2>

<p>{{ post.body }}</p>

</div>

</div>

</template>

export default {

data() {

return {

loading: false,

post: null,

error: null,

}

},

created() {

// watch the params of the route to fetch the data again

this.$watch(

() => this.$route.params,

() => {

this.fetchData()

},

// fetch the data when the view is created and the data is

// already being observed

{ immediate: true }

)

},

methods: {

fetchData() {

this.error = this.post = null

this.loading = true

// replace `getPost` with your data fetching util / API wrapper

getPost(this.$route.params.id, (err, post) => {

this.loading = false

if (err) {

this.error = err.toString()

} else {

this.post = post

}

})

},

},

}

Fetching Before Navigation#

With this approach we fetch the data before actually navigating to the new route. We can perform the data fetching in the beforeRouteEnter guard in the incoming component, and only call next when the fetch is complete:

export default {

data() {

return {

post: null,

error: null,

}

},

beforeRouteEnter(to, from, next) {

getPost(to.params.id, (err, post) => {

next(vm => vm.setData(err, post))

})

},

// when route changes and this component is already rendered,

// the logic will be slightly different.

async beforeRouteUpdate(to, from) {

this.post = null

try {

this.post = await getPost(to.params.id)

} catch (error) {

this.error = error.toString()

}

},

}

The user will stay on the previous view while the resource is being fetched for the incoming view. It is therefore recommended to display a progress bar or some kind of indicator while the data is being fetched. If the data fetch fails, it's also necessary to display some kind of global warning message.

# 第17章 Vue Router and the Composition API

The introduction of setup and Vue's Composition API, open up new possibilities but to be able to get the full potential out of Vue Router, we will need to use a few new functions to replace access to this and in-component navigation guards.

Accessing the Router and current Route inside setup#

Because we don't have access to this inside of setup, we cannot directly access this.$router or this.$route anymore. Instead we use the useRouter function:

import { useRouter, useRoute } from 'vue-router'

export default {

setup() {

const router = useRouter()

const route = useRoute()

function pushWithQuery(query) {

router.push({

name: 'search',

query: {

...route.query,

},

})

}

},

}

The route object is a reactive object, so any of its properties can be watched and you should avoid watching the whole route object. In most scenarios, you should directly watch the param you are expecting to change

import { useRoute } from 'vue-router'

import { ref, watch } from 'vue'

export default {

setup() {

const route = useRoute()

const userData = ref()

// fetch the user information when params change

watch(

() => route.params.id,

async newId => {

userData.value = await fetchUser(newId)

}

)

},

}

Note we still have access to $router and $route in templates, so there is no need to return router or route inside of setup.

Navigation Guards#

While you can still use in-component navigation guards with a setup function, Vue Router exposes update and leave guards as Composition API functions:

import { onBeforeRouteLeave, onBeforeRouteUpdate } from 'vue-router'

import { ref } from 'vue'

export default {

setup() {

// same as beforeRouteLeave option with no access to `this`

onBeforeRouteLeave((to, from) => {

const answer = window.confirm(

'Do you really want to leave? you have unsaved changes!'

)

// cancel the navigation and stay on the same page

if (!answer) return false

})

const userData = ref()

// same as beforeRouteUpdate option with no access to `this`

onBeforeRouteUpdate(async (to, from) => {

// only fetch the user if the id changed as maybe only the query or the hash changed

if (to.params.id !== from.params.id) {

userData.value = await fetchUser(to.params.id)

}

})

},

}

Composition API guards can also be used in any component rendered by <router-view>, they don't have to be used directly on the route component like in-component guards.

useLink#

Vue Router exposes the internal behavior of RouterLink as a Composition API function. It gives access to the same properties as the v-slot API:

import { RouterLink, useLink } from 'vue-router'

import { computed } from 'vue'

export default {

name: 'AppLink',

props: {

// add @ts-ignore if using TypeScript

...RouterLink.props,

inactiveClass: String,

},

setup(props) {

const { route, href, isActive, isExactActive, navigate } = useLink(props)

const isExternalLink = computed(

() => typeof props.to === 'string' && props.to.startsWith('http')

)

return { isExternalLink, href, navigate, isActive }

},

}

# 第18章 Transitions

In order to use transitions on your route components and animate navigations, you need to use the v-slot API:

<router-view v-slot="{ Component }">

<transition name="fade">

<component :is="Component" />

</transition>

</router-view>

All transition APIs work the same here.

Per-Route Transition#

The above usage will apply the same transition for all routes. If you want each route's component to have different transitions, you can instead combine meta fields and a dynamic name on <transition>:

const routes = [

{

path: '/custom-transition',

component: PanelLeft,

meta: { transition: 'slide-left' },

},

{

path: '/other-transition',

component: PanelRight,

meta: { transition: 'slide-right' },

},

]

<router-view v-slot="{ Component, route }">

<!-- Use any custom transition and fallback to `fade` -->

<transition :name="route.meta.transition || 'fade'">

<component :is="Component" />

</transition>

</router-view>

Route-Based Dynamic Transition#

It is also possible to determine the transition to use dynamically based on the relationship between the target route and current route. Using a very similar snippet to the one just before:

<!-- use a dynamic transition name -->

<router-view v-slot="{ Component, route }">

<transition :name="route.meta.transitionName">

<component :is="Component" />

</transition>

</router-view>

We can add an after navigation hook to dynamically add information to the meta field based on the depth of the route

router.afterEach((to, from) => {

const toDepth = to.path.split('/').length

const fromDepth = from.path.split('/').length

to.meta.transitionName = toDepth < fromDepth ? 'slide-right' : 'slide-left'

})

Forcing a transition between reused views#

Vue might automatically reuse components that look alike, avoiding any transition. Fortunately, it is possible to add a key attribute to force transitions. This also allows you to trigger transitions while staying on the same route with different params:

<router-view v-slot="{ Component, route }">

<transition name="fade">

<component :is="Component" :key="route.path" />

</transition>

</router-view>

# 第19章 Scroll Behavior

When using client-side routing, we may want to scroll to top when navigating to a new route, or preserve the scrolling position of history entries just like real page reload does. Vue Router allows you to achieve these and even better, allows you to completely customize the scroll behavior on route navigation.

Note: this feature only works if the browser supports history.pushState.

When creating the router instance, you can provide the scrollBehavior function:

const router = createRouter({

history: createWebHashHistory(),

routes: [...],

scrollBehavior (to, from, savedPosition) {

// return desired position

}

})

The scrollBehavior function receives the to and from route objects, like Navigation Guards. The third argument, savedPosition, is only available if this is a popstate navigation (triggered by the browser's back/forward buttons).

The function can return a ScrollToOptions position object:

const router = createRouter({

scrollBehavior(to, from, savedPosition) {

// always scroll to top

return { top: 0 }

},

})

You can also pass a CSS selector or a DOM element via el. In that scenario, top and left will be treated as relative offsets to that element.

const router = createRouter({

scrollBehavior(to, from, savedPosition) {

// always scroll 10px above the element #main

return {

// could also be

// el: document.getElementById('main'),

el: '#main',

top: -10,

}

},

})

If a falsy value or an empty object is returned, no scrolling will happen.

Returning the savedPosition will result in a native-like behavior when navigating with back/forward buttons:

const router = createRouter({

scrollBehavior(to, from, savedPosition) {

if (savedPosition) {

return savedPosition

} else {

return { top: 0 }

}

},

})

If you want to simulate the "scroll to anchor" behavior:

const router = createRouter({

scrollBehavior(to, from, savedPosition) {

if (to.hash) {

return {

el: to.hash,

}

}

},

})

If your browser supports scroll behavior, you can make it smooth:

const router = createRouter({

scrollBehavior(to, from, savedPosition) {

if (to.hash) {

return {

el: to.hash,

behavior: 'smooth',

}

}

}

})

Delaying the scroll#

Sometimes we need to wait a bit before scrolling in the page. For example, when dealing with transitions, we want to wait for the transition to finish before scrolling. To do this you can return a Promise that returns the desired position descriptor. Here is an example where we wait 500ms before scrolling:

const router = createRouter({

scrollBehavior(to, from, savedPosition) {

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve({ left: 0, top: 0 })

}, 500)

})

},

})

It's possible to hook this up with events from a page-level transition component to make the scroll behavior play nicely with your page transitions, but due to the possible variance and complexity in use cases, we simply provide this primitive to enable specific userland implementations.

# 第20章 Lazy Loading Routes

When building apps with a bundler, the JavaScript bundle can become quite large, and thus affect the page load time. It would be more efficient if we can split each route's components into separate chunks, and only load them when the route is visited.

Vue Router supports dynamic imports out of the box, meaning you can replace static imports with dynamic ones:

// replace

// import UserDetails from './views/UserDetails'

// with

const UserDetails = () => import('./views/UserDetails')

const router = createRouter({

// ...

routes: [{ path: '/users/:id', component: UserDetails }],

})

The component (and components) option accepts a function that returns a Promise of a component and Vue Router will only fetch it when entering the page for the first time, then use the cached version. Which means you can also have more complex functions as long as they return a Promise:

const UserDetails = () =>

Promise.resolve({

/\* component definition \*/

})

In general, it's a good idea to always use dynamic imports for all your routes.

Note

Do not use Async components for routes. Async components can still be used inside route components but route component themselves are just dynamic imports.

When using a bundler like webpack, this will automatically benefit from code splitting

When using Babel, you will need to add the syntax-dynamic-import plugin so that Babel can properly parse the syntax.

Grouping Components in the Same Chunk#

With webpack#

Sometimes we may want to group all the components nested under the same route into the same async chunk. To achieve that we need to use named chunks by providing a chunk name using a special comment syntax (requires webpack > 2.4):

const UserDetails = () =>

import(/\* webpackChunkName: "group-user" \*/ './UserDetails.vue')

const UserDashboard = () =>

import(/\* webpackChunkName: "group-user" \*/ './UserDashboard.vue')

const UserProfileEdit = () =>

import(/\* webpackChunkName: "group-user" \*/ './UserProfileEdit.vue')

webpack will group any async module with the same chunk name into the same async chunk.

With Vite#

In Vite you can define the chunks under the rollupOptions:

// vite.config.js

export default defineConfig({

build: {

rollupOptions: {

// https://rollupjs.org/guide/en/#outputmanualchunks

output: {

manualChunks: {

'group-user': [

'./src/UserDetails',

'./src/UserDashboard',

'./src/UserProfileEdit',

],

},

},

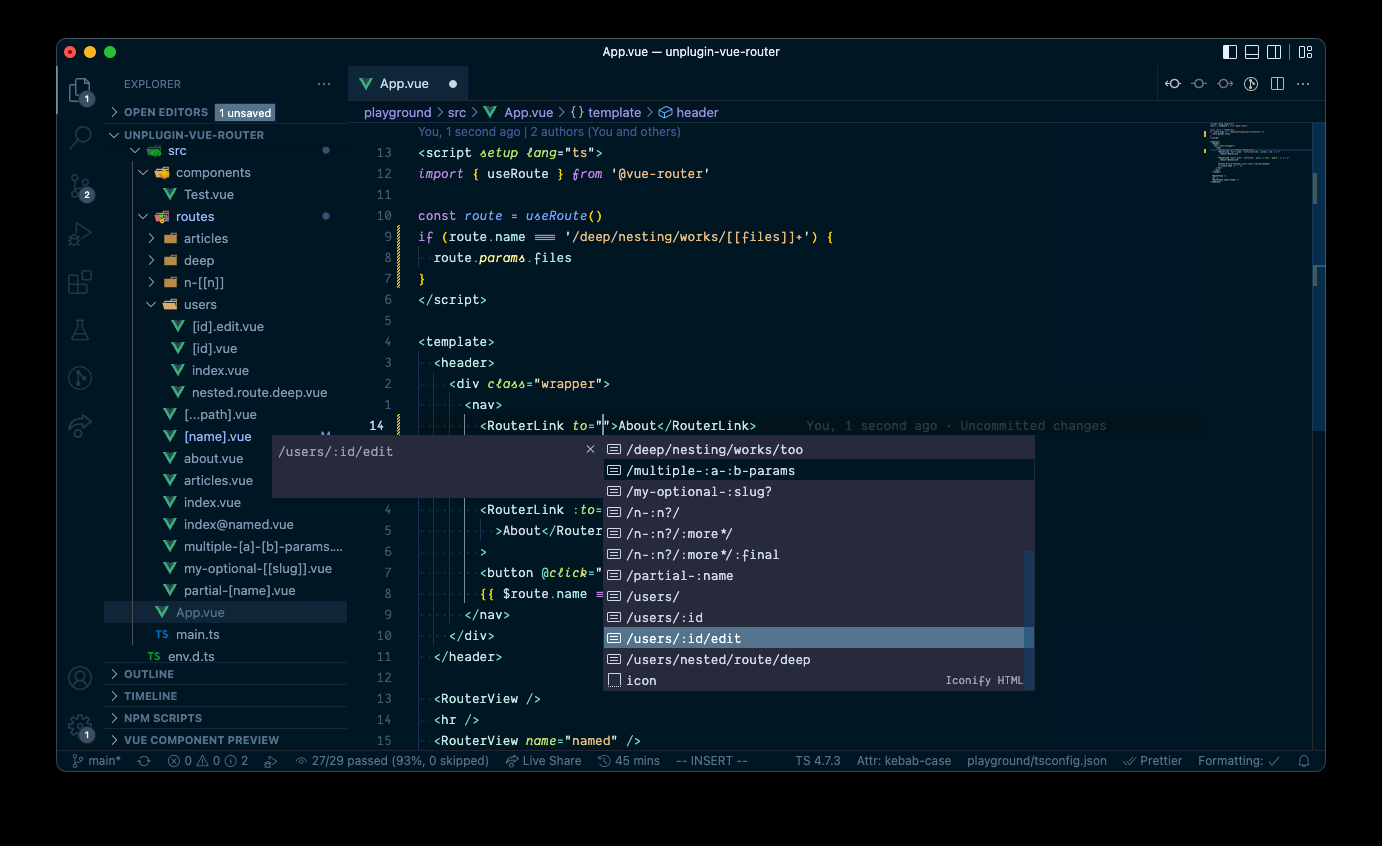
},

})

# 第21章 Typed Routes (v4.1.0+)

‼️ Experimental feature

Starting from v4.1.0, we are introducing a new feature called Typed Routes. This experimental feature is enabled through a Vite/webpack/rollup plugin.



RouterLink to autocomplete

Check the v4.1 release notes for more information about this feature. Check out the plugin GitHub repository for installation instructions and documentation.

# 第22章 Extending RouterLink

The RouterLink component exposes enough props to suffice most basic applications but it doesn't try to cover every possible use case and you will likely find yourself using v-slot for some advanced cases. In most medium to large sized applications, it's worth creating one if not multiple custom RouterLink components to reuse them across your application. Some examples are Links in a Navigation Menu, handling external links, adding an inactive-class, etc.

Let's extend RouterLink to handle external links as well and adding a custom inactive-class in an AppLink.vue file:

<template>

<a v-if="isExternalLink" v-bind="$attrs" :href="to" target="\_blank">

<slot />

</a>

<router-link

v-else

v-bind="$props"

custom

v-slot="{ isActive, href, navigate }"

>

<a

v-bind="$attrs"

:href="href"

@click="navigate"

:class="isActive ? activeClass : inactiveClass"

>

<slot />

</a>

</router-link>

</template>

<script>

import { RouterLink } from 'vue-router'

export default {

name: 'AppLink',

inheritAttrs: false,

props: {

// add @ts-ignore if using TypeScript

...RouterLink.props,

inactiveClass: String,

},

computed: {

isExternalLink() {

return typeof this.to === 'string' && this.to.startsWith('http')

},

},

}

</script>

If you prefer using a render function or create computed properties, you can use the useLink from the Composition API:

import { RouterLink, useLink } from 'vue-router'

export default {

name: 'AppLink',

props: {

// add @ts-ignore if using TypeScript

...RouterLink.props,

inactiveClass: String,

},

setup(props) {

// `props` contains `to` and any other prop that can be passed to <router-link>

const { navigate, href, route, isActive, isExactActive } = useLink(props)

// profit!

return { isExternalLink }

},

}

In practice, you might want to use your AppLink component for different parts of your application. e.g. using Tailwind CSS, you could create a NavLink.vue component with all the classes:

<template>

<AppLink

v-bind="$attrs"

class="inline-flex items-center px-1 pt-1 border-b-2 border-transparent text-sm font-medium leading-5 text-gray-500 focus:outline-none transition duration-150 ease-in-out hover:text-gray-700 hover:border-gray-300 focus:outline-none focus:text-gray-700 focus:border-gray-300 transition duration-150 ease-in-out"

active-class="border-indigo-500 text-gray-900 focus:border-indigo-700"

inactive-class="text-gray-500 hover:text-gray-700 hover:border-gray-300 focus:text-gray-700 focus:border-gray-300"

>

<slot />

</AppLink>

</template>

# 第23 章 Waiting for the result of a Navigation

When using router-link, Vue Router calls router.push to trigger a navigation. While the expected behavior for most links is to navigate a user to a new page, there are a few situations where users will remain on the same page:

Users are already on the page that they are trying to navigate to.

A navigation guard aborts the navigation by doing return false.

A new navigation guard takes place while the previous one not finished.

A navigation guard redirects somewhere else by returning a new location (e.g. return '/login').

A navigation guard throws an Error.

If we want to do something after a navigation is finished, we need a way to wait after calling router.push. Imagine we have a mobile menu that allows us to go to different pages and we only want to hide the menu once we have navigated to the new page, we might want to do something like this:

router.push('/my-profile')

this.isMenuOpen = false

But this will close the menu right away because navigations are asynchronous, we need to await the promise returned by router.push:

await router.push('/my-profile')

this.isMenuOpen = false

Now the menu will close once the navigation is finished but it will also close if the navigation was prevented. We need a way to detect if we actually changed the page we are on or not.

Detecting Navigation Failures#

If a navigation is prevented, resulting in the user staying on the same page, the resolved value of the Promise returned by router.push will be a Navigation Failure. Otherwise, it will be a falsy value (usually undefined). This allows us to differentiate the case where we navigated away from where we are or not:

const navigationResult = await router.push('/my-profile')

if (navigationResult) {

// navigation prevented

} else {

// navigation succeeded (this includes the case of a redirection)

this.isMenuOpen = false

}

Navigation Failures are Error instances with a few extra properties that gives us enough information to know what navigation was prevented and why. To check the nature of a navigation result, use the isNavigationFailure function:

import { NavigationFailureType, isNavigationFailure } from 'vue-router'

// trying to leave the editing page of an article without saving

const failure = await router.push('/articles/2')

if (isNavigationFailure(failure, NavigationFailureType.aborted)) {

// show a small notification to the user

showToast('You have unsaved changes, discard and leave anyway?')

}

TIP

If you omit the second parameter: isNavigationFailure(failure), it will only check if failure is a Navigation Failure.

Differentiating Navigation Failures#

As we said at the beginning, there are different situations aborting a navigation, all of them resulting in different Navigation Failures. They can be differentiated using the isNavigationFailure and NavigationFailureType. There are three different types:

aborted: false was returned inside of a navigation guard to the navigation.

cancelled: A new navigation took place before the current navigation could finish. e.g. router.push was called while waiting inside of a navigation guard.

duplicated: The navigation was prevented because we are already at the target location.

Navigation Failures's properties#

All navigation failures expose to and from properties to reflect the current location as well as the target location for the navigation that failed:

// trying to access the admin page

router.push('/admin').then(failure => {

if (isNavigationFailure(failure, NavigationFailureType.redirected)) {

failure.to.path // '/admin'

failure.from.path // '/'

}

})

In all cases, to and from are normalized route locations.

Detecting Redirections#

When returning a new location inside of a Navigation Guard, we are triggering a new navigation that overrides the ongoing one. Differently from other return values, a redirection doesn't prevent a navigation, it creates a new one. It is therefore checked differently, by reading the redirectedFrom property in a Route Location:

await router.push('/my-profile')

if (router.currentRoute.value.redirectedFrom) {

// redirectedFrom is resolved route location like to and from in navigation guards

}

# 第24章 Dynamic Routing

Adding routes to your router is usually done via the routes option but in some situations, you might want to add or remove routes while the application is already running. Application with extensible interfaces like Vue CLI UI can use this to make the application grow.

Adding Routes#

Dynamic routing is achieved mainly via two functions: router.addRoute() and router.removeRoute(). They only register a new route, meaning that if the newly added route matches the current location, it would require you to manually navigate with router.push() or router.replace() to display that new route. Let's take a look at an example:

Imagine having the following router with one single route:

const router = createRouter({

history: createWebHistory(),

routes: [{ path: '/:articleName', component: Article }],

})

Going to any page, /about, /store, or /3-tricks-to-improve-your-routing-code ends up rendering the Article component. If we are on /about and we add a new route:

router.addRoute({ path: '/about', component: About })

The page will still show the Article component, we need to manually call router.replace() to change the current location and overwrite where we were (instead of pushing a new entry, ending up in the same location twice in our history):

router.addRoute({ path: '/about', component: About })

// we could also use this.$route or route = useRoute() (inside a setup)

router.replace(router.currentRoute.value.fullPath)

Remember you can await router.replace() if you need to wait for the new route to be displayed.

Adding Routes inside navigation guards#

If you decide to add or remove routes inside of a navigation guard, you should not call router.replace() but trigger a redirection by returning the new location:

router.beforeEach(to => {

if (!hasNecessaryRoute(to)) {

router.addRoute(generateRoute(to))

// trigger a redirection

return to.fullPath

}

})

The example above assumes two things: first, the newly added route record will match the to location, effectively resulting in a different location from the one we were trying to access. Second, hasNecessaryRoute() returns false after adding the new route to avoid an infinite redirection.

Because we are redirecting, we are replacing the ongoing navigation, effectively behaving like the example shown before. In real world scenarios, adding is more likely to happen outside of navigation guards, e.g. when a view component mounts, it register new routes.

Removing routes#

There are few different ways to remove existing routes:

By adding a route with a conflicting name. If you add a route that has the same name as an existing route, it will remove the route first and then add the route:

router.addRoute({ path: '/about', name: 'about', component: About })

// this will remove the previously added route because they have the same name and names are unique

router.addRoute({ path: '/other', name: 'about', component: Other })

By calling the callback returned by router.addRoute():

const removeRoute = router.addRoute(routeRecord)

removeRoute() // removes the route if it exists

This is useful when the routes do not have a name

By using router.removeRoute() to remove a route by its name:

router.addRoute({ path: '/about', name: 'about', component: About })

// remove the route

router.removeRoute('about')

Note you can use Symbols for names in routes if you wish to use this function but want to avoid conflicts in names.

Whenever a route is removed, all of its aliases and children are removed with it.

Adding nested routes#

To add nested routes to an existing route, you can pass the name of the route as its first parameter to router.addRoute(), this will effectively add the route as if it was added through children:

router.addRoute({ name: 'admin', path: '/admin', component: Admin })

router.addRoute('admin', { path: 'settings', component: AdminSettings })

This is equivalent to:

router.addRoute({

name: 'admin',

path: '/admin',

component: Admin,

children: [{ path: 'settings', component: AdminSettings }],

})

Looking at existing routes#

Vue Router gives you two functions to look at existing routes:

router.hasRoute(): check if a route exists

router.getRoutes(): get an array with all the route records.