Protecting Vue Routes with Navigation Guards

Author
Divya Sasidharan

1 Comment
Join the Conversation →

Published
Jul 11, 2019

Updated
Jul 11, 2019

AUTHENTICATION IDENTITY VUE VUE-ROUTER

Authentication is a necessary part of every web application. It is a handy means by which we can personalize experiences and load content specific to a user — like a logged in state. It can also be used to evaluate permissions, and prevent otherwise private information from being accessed by unauthorized users.

A common practice that applications use to protect content is to house them under specific routes and build redirect rules that navigate users toward or away from a resource depending on their permissions. To gate content reliably behind protected routes, they need to build to separate static pages. This way, redirect rules can properly handle redirects.

In the case of Single Page Applications (SPAs) built with modern front-end frameworks, like Vue, redirect rules cannot be utilized to protect routes. Because all pages are served from a single entry file, from a browser's perspective, there is only one page: `index.html`. In a SPA, route logic generally stems from a routes file. This is where we will do most of our auth configuration for this post. We will specifically lean on Vue's navigation guards to handle authentication specific routing since this helps us access selected routes before it fully resolves. Let's dig in to see how this works.

Roots and Routes

Navigation guards are a specific feature within Vue Router that provide additional functionality pertaining to how routes get resolved. They are primarily used to handle error states and navigate a user seamlessly without abruptly interrupting their workflow.

There are three main categories of guards in Vue Router: Global Guards, Per Route Guards and In Component Guards. As the names suggest, **Global Guards** are called when any navigation is triggered (i.e. when URLs change), **Per Route Guards** are called when the associated route is called (i.e. when a URL matches a specific route), and **Component Guards** are called when a component in a route is created, updated or destroyed. Within each category, there are additional methods that gives you more fine grained control of application routes. Here's a quick break down of all available methods within each type of navigation guard in Vue Router.

Global Guards

- **`beforeEach`**: action before entering any route (no access to `this` scope)
- **`beforeResolve`**: action before the navigation is confirmed, but after in-component guards (same as beforeEach with `this` scope access)
- **`afterEach`**: action after the route resolves (cannot affect navigation)

Per Route Guards

- **`beforeEnter`**: action before entering a specific route (unlike global guards, this has access to `this`)

Component Guards

- **`beforeRouteEnter`**: action before navigation is confirmed, and before component creation (no access to this)
- **`beforeRouteUpdate`**: action after a new route has been called that uses the same component
- **`beforeRouteLeave`**: action before leaving a route

Protecting Routes

To implement them effectively, it helps to know when to use them in any given scenario. If you wanted to track page views for analytics for instance, you may want to use the global `afterEach` guard, since it gets fired when the route and associated components are fully resolved. And if you wanted to prefetch data to load onto a Vuex store before a route resolves, you could do so using the `beforeEnter` per route guard.

Since our example deals with protecting specific routes based on a user's access permissions, we will use **in component** navigation guards, namely the `beforeEnter` hook. This navigation guard gives us access to the proper route before the resolve completes; meaning that we can fetch data or check that data has loaded before letting a user pass through. Before diving into the implementation details of how this works, let's briefly look at how our `beforeEnter` hook fits into our existing routes file. Below, we have our sample routes file, which has our protected route, aptly named `protected`. To this, we will add our `beforeEnter` hook to it like so:

JavaScript

```
const router = new VueRouter({
  routes: [
    ...
    {
      path: "/protected",
      name: "protected",
      component: import(/* webpackChunkName: "protected" */ './Protected.vue'),
      beforeEnter(to, from, next) {
        // logic here
      }
  ]
})
```

Anatomy of a route

The anatomy of a `beforeEnter` is not much different from other available navigation guards in Vue Router. It accepts three parameters: **`to`**, the "future" route the app is navigating to; **`from`**, the "current/soon past" route the app is navigating away from and **`next`**, a function that must be called for the route to resolve successfully.

Generally, when using Vue Router, `next` is called without any arguments. However, this assumes a perpetual success state. In our case, we want to ensure that unauthorized users who fail to enter a protected resource have an alternate path to take that redirects them appropriately. To do this, we will pass in an argument to `next`. For this, we will use the name of the route to navigate users to if they are unauthorized like so:

JavaScript

```
next({
  name: "dashboard"
})
```

Let's assume in our case, that we have a Vuex store where we store a user's authorization token. In order to check that a user has permission, we will check this store and either fail or pass the route appropriately.

JavaScript

```
beforeEnter(to, from, next) {
  // check vuex store //
  if (store.getters["auth/hasPermission"]) {
    next()
  } else {
    next({
      name: "dashboard" // back to safety route //
    });
  }
}
```

In order to ensure that events happen in sync and that the route doesn't prematurely load before the Vuex action is completed, let's convert our navigation guards to use async/await.

JavaScript

```
async beforeEnter(to, from, next) {
  try {
    var hasPermission = await store.dispatch("auth/hasPermission");
    if (hasPermission) {
      next()
    }
  } catch (e) {
    next({
      name: "dashboard" // back to safety route //
    })
  }
}
```

Never forget where you came from

So far our navigation guard fulfills its purpose of preventing unauthorized users access to protected resources by redirecting them to where they may have come from (i.e. the dashboard page). Even so, such a workflow is disruptive. Since the redirect is unexpected, a user may assume user error and attempt to access the route repeatedly with the eventual assumption that the application is broken. To account for this, let's create a way to let users know when and why they are being redirected.

We can do this by passing in a query parameter to the `next` function. This allows us to append the protected resource path to the redirect URL. So, if you want to prompt a user to log into an application or obtain the proper permissions without having to remember where they left off, you can do so. We can get access to the path of the protected resource via the `to` route object that is passed into the `beforeEnter` function like so: `to.fullPath`.

JavaScript

```
async beforeEnter(to, from, next) {
  try {
    var hasPermission = await store.dispatch("auth/hasPermission");
    if (hasPermission) {
      next()
    }
  } catch (e) {
    next({
      name: "login", // back to safety route //
      query: { redirectFrom: to.fullPath }
    })
  }
}
```

Notifying

The next step in enhancing the workflow of a user failing to access a protected route is to send them a message letting them know of the error and how they can solve the issue (either by logging in or obtaining the proper permissions). For this, we can make use of in component guards, specifically, `beforeRouteEnter`, to check whether or not a redirect has happened. Because we passed in the redirect path as a query parameter in our routes file, we now can check the route object to see if a redirect happened.

JavaScript

```
beforeRouteEnter(to, from, next) {
  if (to.query.redirectFrom) {
    // do something //
  }
}
```

As I mentioned earlier, all navigation guards must call `next` in order for a route to resolve. The upside to the `next` function as we saw earlier is that we can pass an object to it. What you may not have known is that you can also access the Vue instance within the next function. Wuuuuuuut? Here's what that looks like:

JavaScript

```
next(() => {
  console.log(this) // this is the Vue instance
})
```

You may have noticed that you don't *technically* have access to the `this` scope when using `beforeEnter`. Though this might be the case, you can still access the Vue instance by passing in the `vm` to the function like so:

JavaScript

```
next(vm => {
  console.log(vm) // this is the Vue instance
})
```

This is especially handy because you can now create and appropriately update a data property with the relevant error message when a route redirect happens. Say you have a data property called `errorMsg`. You can now update this property from the `next` function within your navigation guards easily and without any added configuration. Using this, you would end up with a component like this:

HTML

```
<template>
  <div>
    <span>{{ errorMsg }}</span>
    <!-- some other fun content -->
    ...
    <!-- some other fun content -->
  </div>
</template>
```

JavaScript

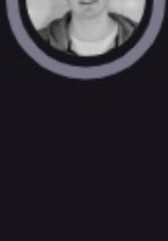
```
<script>
export default {
  name: "Error",
  data() {
    return {
      errorMsg: null
    }
  },
  beforeRouteEnter(to, from, next) {
    if (to.query.redirectFrom) {
      next(vm => {
        vm.errorMsg =
          "Sorry, you don't have the right access to reach the route requested"
      })
    } else {
      next()
    }
  }
}
</script>
```

Conclusion

The process of integrating authentication into an application can be a tricky one. We covered how to gate a route from unauthorized access as well as how to put workflows in place that redirect users toward and away from a protected resource based on their permissions. The assumption thus far has been that you already have authentication configured in your application. If you don't yet have this configured and you'd like to get up and running fast, I highly recommend working with authentication as a service. There are providers like Netlify's Identity Widget or Auth0's lock.

Share: Twitter Facebook

Comments



Max
July 11, 2019

Hey,

great article, thank you very much. I think I found a small mistake in this paragraph:

"Since our example deals with protecting specific routes based on a user's access permissions, we will use *in component* navigation guards"

you say you're using "in component" guards but then you go ahead and use `beforeEnter` which you previously specified as "per route guards".

Have the best day. :)