Handling Authentication In Vue Using Vuex

While this tutorial has content that we believe is of great benefit to our community, we have not yet tested or edited it to ensure you have an error-free learning experience. It's on our list, and we're working on it! You can help us out by using the "report an issue" button at the bottom of the tutorial.

Introduction

Traditionally, many people use local storage to manage tokens generated through client-side authentication. A big concern is always a better way to manage authorization tokens to allow us to store even more information on users.

This is where Vuex comes in. Vuex manages states for Vue.js applications. It serves as a centralized store for all the components in an application, with rules ensuring that the state can only be mutated in a predictable fashion.

Sounds like a better alternative to always checking localStorage? Let’s explore it.

Prerequisites

1. Node installed on your local system
2. Knowledge of JavaScript and Vue
3. Install Vue CLI on your local system.
4. Read through Vue Authentication And Route Handling Using Vue-router

If you want to jump straight to the demo code: **Go to `vue-auth-vuex` on GitHub**

Setting up the application modules

For this project, we want to create a vue application that has vuex and vue-router. We will use the vue cli 3.0 to create a new vue project and select router and vuex from the options.

Run the following command to set it up:

```
$ vue create vue-auth
```

Follow the dialogue that shows up, add the necessary information and select the options we need and complete the installation.

Next, install axios:

```
$ npm install axios --save
```

Setup Axios

We will need axios across many of our components. Let's set it up at the entry level so we do not have to import it every time we need it.

Open the `./src/main.js` file and add the following:

```
[...]
import store from './store'
import Axios from 'axios'

Vue.prototype.$http = Axios;
const token = localStorage.getItem('token')
if (token) {
  Vue.prototype.$http.defaults.headers.common['Authorization'] = token
}
[...]
```

Now, when we want to use axios inside our component, we can do `this.$http` and it will be like calling axios directly. We also set the `Authorization` on axios header to our token, so our requests can be processed if a token is required. This way, we do not have to set token anytime we want to make a request.

When that is done, let's set up the server to handle authentication.

Setting up the server for authentication

I already wrote about this when explaining how to handle authentication with vue-router. Check out the Setup Node.js Server section of this

Setup Components

The Login Component

Create a file `Login.vue` in the `./src/components` directory. Then, add the template for the login page:

```
<template>
 <div>
   <form class="login" @submit.prevent="login">
     <h1>Sign in</h1>
     <label>Email</label>
     <input required v-model="email" type="email" placeholder="Name"/>
     <label>Password</label>
     <input required v-model="password" type="password" placeholder="Password"/>
     <hr/>
     <button type="submit">Login</button>
   </form>
 </div>
</template>
```

When you are done, add the data attributes that would bind to the HTML form:

```
[...]
<script>
  export default {
    data(){
      return {
        email : "",
        password : ""
      }
    },
  }
</script>
```

Now, let's add the method for handling login:

```
[...]
<script>
  export default {
    [...]
    methods: {
      login: function () {
        let email = this.email
        let password = this.password
        this.$store.dispatch('login', { email, password })
       .then(() => this.$router.push('/'))
       .catch(err => console.log(err))
      }
    }
  }
</script>
```

We are using a vuex action — `login` to handle this authentication. We can resolve actions into promises so we can do cool things with them inside our component.

The Register Component

Like the component for login, let's make one for registering users. Start by creating a file `Register.vue` in the components directory and add the following to it:

```
<template>
  <div>
    <h4>Register</h4>
    <form @submit.prevent="register">
      <label for="name">Name</label>
      <div>
          <input id="name" type="text" v-model="name" required autofocus>
      </div>

      <label for="email" >E-Mail Address</label>
      <div>
          <input id="email" type="email" v-model="email" required>
      </div>

      <label for="password">Password</label>
      <div>
          <input id="password" type="password" v-model="password" required>
      </div>

      <label for="password-confirm">Confirm Password</label>
      <div>
          <input id="password-confirm" type="password" v-
model="password_confirmation" required>
      </div>

      <div>
          <button type="submit">Register</button>
      </div>
    </form>
  </div>
</template>
```

Let define the data attributes we will bind to the form:

```
[...]
<script>
  export default {
    data(){
      return {
        name : "",
        email : "",
        password : "",
        password_confirmation : "",
        is_admin : null
      }
    },
  }
</script>
```

Now, let’s add the method for handling login:

```
[...]
<script>
  export default {
    [...]
    methods: {
      register: function () {
        let data = {
          name: this.name,
          email: this.email,
          password: this.password,
          is_admin: this.is_admin
        }
        this.$store.dispatch('register', data)
       .then(() => this.$router.push('/'))
       .catch(err => console.log(err))
      }
    }
  }
</script>
```

The Secure Component

Let's make a simple component that would only display if our user is authenticated. Create the component file `Secure.vue` and add the following to it:

```
<template>
  <div>
    <h1>This page is protected by auth</h1>
  </div>
</template>
```

Update The App Component

Open `./src/App.vue` file and add the following to it:

```
<template>
  <div id="app">
    <div id="nav">
      <router-link to="/">Home</router-link> |
      <router-link to="/about">About</router-link><span v-if="isLoggedIn"> | <a
@click="logout">Logout</a></span>
    </div>
    <router-view/>
  </div>
</template>
```

Can you see the `Logout` link we set to only show up if a user is logged in? Great.

Now, let's add the logic behind the log out:

```
<script>
  export default {
    computed : {
      isLoggedIn : function(){ return this.$store.getters.isLoggedIn}
    },
    methods: {
      logout: function () {
        this.$store.dispatch('logout')
        .then(() => {
          this.$router.push('/login')
        })
      }
    },
  }
</script>
```

We are doing two things — computing the authentication state of the user and dispatching a logout action to our vuex store when a user clicks the logout button. After the log out, we send the user to `login` page using `this.$router.push('/login')`. You can change where the user gets sent to if you want.

That's it. Let's make the auth module using vuex.

Vuex Auth Module

If you read past the Setup Node.js Server section, you would notice we had to store user auth token in localStorage and we had to retrieve both the token and user information anytime we wanted to check if the user is authenticated. This works, but it is not really elegant. We will rebuild the authentication to use vuex.

First, let's setup our `store.js` file for vuex:

```
import Vue from 'vue'
import Vuex from 'vuex'
import axios from 'axios'

Vue.use(Vuex)

export default new Vuex.Store({
  state: {
    status: '',
    token: localStorage.getItem('token') || '',
    user : {}
  },
  mutations: {

  },
  actions: {

  },
  getters : {

  }
})
```

If you noticed, we have imported vue, vuex and axios, then asked vue to use vuex. This is because we mean serious business here.

We have defined the attributes of the state. Now the vuex state would hold our authentication status, `jwt` token and user information.

Create The Vuex `login` Action

Vuex actions are used to commit mutations to the vuex store. We will create a `login` action that would authenticate a user with the server and commit user credentials to the vuex store. Open the `./src/store.js` file and add the following to actions object:

```
login({commit}, user){
    return new Promise((resolve, reject) => {
      commit('auth_request')
      axios({url: 'http://localhost:3000/login', data: user, method: 'POST' })
      .then(resp => {
        const token = resp.data.token
        const user = resp.data.user
        localStorage.setItem('token', token)
        axios.defaults.headers.common['Authorization'] = token
        commit('auth_success', token, user)
        resolve(resp)
      })
      .catch(err => {
        commit('auth_error')
        localStorage.removeItem('token')
        reject(err)
      })
    })
},
```

The login action passes vuex `commit` helper that we will use to trigger mutations. Mutations make changes to vuex store.

We are making a call to the server's login route and returning the necessary data. We store the token on localStorage, then pass the token and user information to `auth_success` to update the store's attributes. We also set the header for `axios` at this point as well.

> We could store the token in vuex store, but if the user leaves our application, all of the data in the vuex store disappears. To ensure we allow the user to return to the application within the validity time of the token and not have to log in again, we have to keep the token in localStorage.
>
> It's important you know how these work so you can decide what exactly it is you want to achieve.

We return a promise so we can return a response to a user after login is complete.

Create The Vuex `register` Action

Like the `login` action, the `register` action will work almost the same way. In the same file, add the following in the actions object:

```
register({commit}, user){
  return new Promise((resolve, reject) => {
    commit('auth_request')
    axios({url: 'http://localhost:3000/register', data: user, method: 'POST' })
    .then(resp => {
      const token = resp.data.token
      const user = resp.data.user
      localStorage.setItem('token', token)
      axios.defaults.headers.common['Authorization'] = token
      commit('auth_success', token, user)
      resolve(resp)
    })
    .catch(err => {
      commit('auth_error', err)
      localStorage.removeItem('token')
      reject(err)
    })
  })
},
```

This works similarly to `login` action, calling the same mutators as our `login` and `register` actions have the same simple goal — get a user into the system.

Create The Vuex `logout` Action

We want the user to have the ability to log out of the system, and we want to destroy all data created during the last authenticated session. In the same `actions` object, add the following:

```
logout({commit}){
  return new Promise((resolve, reject) => {
    commit('logout')
    localStorage.removeItem('token')
    delete axios.defaults.headers.common['Authorization']
    resolve()
  })
}
```

Now, when the user clicks to log out, we will remove the `jwt` token we stored along with the `axios` header we set. There is no way they can perform a transaction requiring a token now.

Create The Mutations

Like I mentioned earlier, mutators are used to change the state of a vuex store. Let's define the mutators we had used throughout our application. In the mutators object, add the following:

```
mutations: {
  auth_request(state){
    state.status = 'loading'
  },
  auth_success(state, token, user){
    state.status = 'success'
    state.token = token
    state.user = user
  },
  auth_error(state){
    state.status = 'error'
  },
  logout(state){
    state.status = ''
    state.token = ''
  },
},
```

Create The Getters

We use getter to get the value of the attributes of vuex state. The role of our getter in the situation is to separate application data from application logic and ensure we do not give away sensitive information.

Add the following to the `getters` object:

```
getters : {
  isLoggedIn: state => !!state.token,
  authStatus: state => state.status,
}
```

You would agree with me that this is a neater way to access data in the store.

Hide Pages Behind Auth

The whole purpose of this article is to implement authentication and keep certain pages away from a user who is not authentication. To achieve this, we need to know the page the user wants to visit and equally have a way to check if the user is authenticated. We also need a way to say if the page is reserved for only authenticated user or unauthenticated user alone or both. These things are important considerations which, luckily, we can achieve with vue-router.

Defiing Routes For Authenticated And Unauthenticated Pages

Open the `./src/router.js` file and import what we need for this setup:

```
import Vue from 'vue'
import Router from 'vue-router'
import store from './store.js'
import Home from './views/Home.vue'
import About from './views/About.vue'
import Login from './components/Login.vue'
import Secure from './components/Secure.vue'
import Register from './components/Register.vue'

Vue.use(Router)
```

As you can see, we have imported vue, vue-router and our vuex store setup. We also imported all the components we defined and set vue to use our router.

Let's define the routes:

```
[...]
let router = new Router({
  mode: 'history',
  routes: [
    {
      path: '/',
      name: 'home',
      component: Home
    },
    {
      path: '/login',
      name: 'login',
      component: Login
    },
    {
      path: '/register',
      name: 'register',
      component: Register
    },
    {
      path: '/secure',
      name: 'secure',
      component: Secure,
      meta: {
        requiresAuth: true
      }
    },
    {
      path: '/about',
      name: 'about',
      component: About
    }
  ]
})

export default router
```

Our route definition is simple. For routes requiring authentication, we add extra data to it to enable us identify it when the user tries to access it. This is the essence of the `meta` attribute added to the route definition. If you are asking ***"Can I add more data to this*** `meta` ***and use it?"*** then I'm pleased to tell you that you are absolutely right ?.

Handling Unauthorized Access Cases

We have our routes defined. Now, let's check for unauthorized access and take action. In the `router.js` file, add the following before the `export default router`:

```
router.beforeEach((to, from, next) => {
  if(to.matched.some(record => record.meta.requiresAuth)) {
    if (store.getters.isLoggedIn) {
      next()
      return
    }
    next('/login')
  } else {
    next()
  }
})
```

From the article on using vue router for authentication, you can recall we had a really complex mechanism here that grew very big and got very confusing. Vuex has helped us simplify that completely, and we can go on to add any condition to our route. In our vuex store, we can then define actions to check these conditions and getters to return them.

Handling Expired Token Cases

Because we store our token in localStorage, it can remain there perpetually. This means that whenever we open our application, it would automatically authenticate a user even if the token has expired. What would happen at most is that our requests would keep failing because of an invalid token. This is bad for user experience.

Now, open `./src/App.vue` file and in the script, add the following to it:

```
export default {
  [...]
  created: function () {
    this.$http.interceptors.response.use(undefined, function (err) {
      return new Promise(function (resolve, reject) {
        if (err.status === 401 && err.config && !err.config.__isRetryRequest) {
          this.$store.dispatch(logout)
        }
        throw err;
      });
    });
  }
}
```

We are intercepting axios call to determine if we get `401 Unauthorized` response. If we do, we dispatch the `logout` action and the user gets logged out of the application. This takes them to the `login` page like we designed earlier and they can log in again.

We can agree that this will greatly improve the user's experience.

Conclusion

Using vuex allows us to store and manage authentication state and proceed to check state in our application using only a few lines of code.

- Comments
- Follow-Up Questions