

6.1040 Rec 9

Vuex and Vue Router



What we'll be covering today

- State management with Vuex
 - Mutations
 - Getters
- Server-side vs. client-side routing
- Vue Router
 - Dynamic routing
 - Programmatic navigation
 - Navigation guards

Motivation for Vuex

- Passing data between different components can be difficult
 - Tedious for deeply nested components
 - Straight up doesn't work for sibling components
- Unsustainable workarounds:
 - Reaching for direct parent/child instance references
 - Trying to mutate and synchronize multiple copies of the state via events

Motivation for Vuex

- Instead, extract the shared state out of the components into a centralized **store**
 - Ensures the state can only be mutated in a predictable fashion
 - Any component can access the state or trigger actions, no matter where they are in the tree!
- We can accomplish this in Vue through the [Vuex library](#)
 - Vuex uses a **single state tree**
 - Great for keeping track of variables "globally"

Basic Vuex usage

- In the store, define:
 - Initial values of the **state**
 - Synchronous **mutations** to modify the **state**
- Inject store into root component
- Accessing store from child components: **this.\$store**
 - Call **commit()** with mutation name to trigger mutation
 - Access state values from **.state**

```
1 // store.ts
2 import Vue from 'vue'
3 import Vuex from 'vuex'
4
5 Vue.use(Vuex)
6
7 const store = new Vuex.Store({
8   state: {
9     count: 0
10  },
11  mutations: {
12    increment(state) {
13      state.count++
14    }
15  }
16 })
17
18 new Vue({ store }).$mount('#app');
19
20 // ChildComponent.vue
21 methods: {
22   increment() {
23     this.$store.commit('increment')
24     console.log(this.$store.state.count)
25   }
26 }
```

Exercise: Todo List

<https://github.com/61040-fa22/rec9>

Demo at /public/demo.html

Exercise 1

- Move todo items into the Vuex store
 - Add a new state variable **items**
 - Add a new mutation **addItem**
 - Update **TodoInputForm** and **TodoListPage** to use the store instead of keeping data and emitting events to **TodoListPage**

```
1 // store.ts
2 import Vue from 'vue'
3 import Vuex from 'vuex'
4
5 Vue.use(Vuex)
6
7 const store = new Vuex.Store({
8   state: {
9     count: 0
10  },
11  mutations: {
12    increment(state) {
13      state.count++
14    }
15  }
16 })
17
18 new Vue({ store }).$mount('#app');
19
20 // ChildComponent.vue
21 methods: {
22   increment() {
23     this.$store.commit('increment')
24     console.log(this.$store.state.count)
25   }
26 }
```

Store getters

- Help dynamically compute values from store values
- In the store:
 - Getters receive the **state** as their 1st argument for you to compute values off of
 - You can also pass arguments to getters by returning a function
- The getters will be exposed on the **store.getters** for you to access

```
1 // store.ts
2 const store = new Vuex.Store({
3   state: {
4     products: [
5       { name: 'coffee', inStock: true },
6       { name: 'tea', inStock: false }
7     ]
8   },
9   getters: {
10    inStockProducts: state => {
11      return state.products
12        .filter(p => p.inStock)
13    },
14    getProductByName: (state) => (name) => {
15      return state.products
16        .find(p => p.name === name)
17    }
18  }
19 });
20
21 // ChildComponent.vue:
22 computed: {
23   itemsInStock() {
24     return this.$store.getters
25       .inStockProducts.join(', ');
26   },
27   isTeaInStock() {
28     return this.$store.getters
29       .getProductByName('tea').inStock;
30   }
31 }
```


Exercise 2a

- Make a new page **TodoStatsPage** that displays:
 - The total number of items in the todo list
 - A new store getter that finds the number of todo items with keyword “important” in them
 - Your answer should be just a few lines! No need for imports, data(), methods() etc.

```
9  getters: {
10   inStockProducts: state => {
11     return state.products
12       .filter(p => p.inStock)
13   },
14   getProductByName: (state) => (name) => {
15     return state.products
16       .find(p => p.name === name)
17   }
18 }
19 });
20
```

Client vs. server-side routing

- Server-side routing (traditional method)
 - Browser requests new page content from web server
 - When the server responds with HTML content, **the entire page reloads to render the new content**
- Client-side routing (new method)
 - Intercept user navigation to a different page on the website
 - Dynamically fetch template data to update the view **without reloading page**
 - **Problem:** How do we update browser history if we technically stay on the same page the whole time?

Vue Router

- Maps specific components to be displayed in the router-view depending on the path you're visiting
 - For example, a component mapped to route **/docs** would be displayed `vuejs.org/#/docs`
- Provides convenient ways to change the user's navigation programmatically
- Tracks browser navigation and history under the hood

Basic router usage

- Define view components, or import them from other files.
- Define **routes**, each an object with the following options:
 - **path** to render the component at
 - **component** to render
 - **name** for the route
- Create a **router** instance
 - Pass in the **routes** as a param option
- Mount the **router** onto the root component

```
1 // router.ts
2 import Vue from 'vue';
3 import VueRouter from 'vue-router';
4 import TodoListPage from './TodoList/TodoListPage.vue';
5
6 Vue.use(VueRouter);
7
8 const routes = [
9   {path: '/', name: 'Home', component: TodoListPage},
10 ];
11
12 const router = new VueRouter({routes});
13
14 new Vue({ router }).$mount('#app');
```

Basic router usage

- Page component matching route will replace the **<router-view>** element
- **<router-link>** enables navigation in a router-enabled app
 - It renders as an **<a>** tag with the specified **href** by default
 - Automatically gets an active CSS class when the target route is active
 - The target location is specified with the **to** prop

```
1 // App.vue
2 <template>
3   <div id="app">
4     <header>
5       <router-link to="/">
6         Home
7       </router-link>
8     </header>
9     <router-view />
10  </div>
11 </template>
```

Exercise 2b

- Modify the router to map the **TodoStatsPage** component you made in 2a to the route `"/stats"`

```
1 // router.ts
2 import Vue from 'vue';
3 import VueRouter from 'vue-router';
4 import TodoListPage from './TodoList/TodoListPage.vue';
5
6 Vue.use(VueRouter);
7
8 const routes = [
9   {path: '/', name: 'Home', component: TodoListPage},
10 ];
11
12 const router = new VueRouter({routes});
13
14 new Vue({ router }).$mount('#app');
```

```
1 // App.vue
2 <template>
3   <div id="app">
4     <header>
5       <router-link to="/">
6         Home
7       </router-link>
8     </header>
9     <router-view />
10  </div>
11 </template>
```

Dynamic routing

- Some routes with a common patterns should map to the same component
 - E.g. using the same Profile component to render a user profile for each user, but with different user IDs
- We can use a **dynamic segment** in the path to achieve this
 - Access the named segment from **\$route.params**

pattern	matched path	\$route.params
/user/:username	/user/evan	<pre>{ username: 'evan' }</pre>
/user/:username /post/:post_id	/user/evan /post/123	<pre>{ username: 'evan', post_id: '123' }</pre>

Exercise 3

- Add a new **TodoFilterStatsPage** with the following details:
 - Maps to route **/stats/SOMEKEYWORDHERE**
 - Given the keyword, shows the total number of items in the todo list containing the keyword (store getter)
 - Again, the new page should only be a few lines

pattern	matched path	\$route.params
/user/:username	/user/evan	<pre>{ username: 'evan' }</pre>
/user/:username /post/:post_id	/user/evan /post/123	<pre>{ username: 'evan', post_id: '123' }</pre>

Programmatic Navigation

- To programmatically change what page gets rendered at any time, access the router through **this.\$router**
 - Navigate to different URL: **router.push()**
 - Navigate to different URL without updating history: **router.replace()**
 - Navigate to previous page in history: **router.go(-1)**
 - Can move to the previous n-th page with **router.go(-n)**
 - Can move to the next n-th page with **router.go(n)**
- **Check your understanding:**
 - Why don't we use **<router-link>** in authentication forms and instead rely on programmatic navigation?
 - Where in the starter code do we use the router to programmatically change the route?

Exercise 4

- Add some tools to help user navigate to and from the keyword stats page:
 - Add a new **TodoFilterForm** programmatically navigating to the corresponding filter page when they submit the form
 - **Note:** Make a copy of **TodoInputForm** and modify accordingly
 - Add a Back button on the **TodoFilterStatsPage** to allow user to return to their previous page

Navigation guards

- We can prevent users from accessing pages they're not supposed to visit with **navigation guards**
- Use **router.beforeEach** to check if some page the user is attempting to navigate **to from** a different page, is acceptable
- Specify the **next()** function to control if they can proceed as desired, or redirect them to an entirely different page (similar to Express!)

```
1 // lines 22-36 in router.ts
2 router.beforeEach((to, from, next) => {
3   if (router.app.$store) {
4     if (to.name === 'Login' && router.app.$store.state.username) {
5       next({name: 'Account'}); // Go to Account page if user navigates to Login and are signed in
6       return;
7     }
8
9     if (to.name === 'Account' && !router.app.$store.state.username) {
10      next({name: 'Login'}); // Go to Login page if user navigates to Account and are not signed in
11      return;
12    }
13  }
14
15  next();
16 });
```