Advanced Web Programming

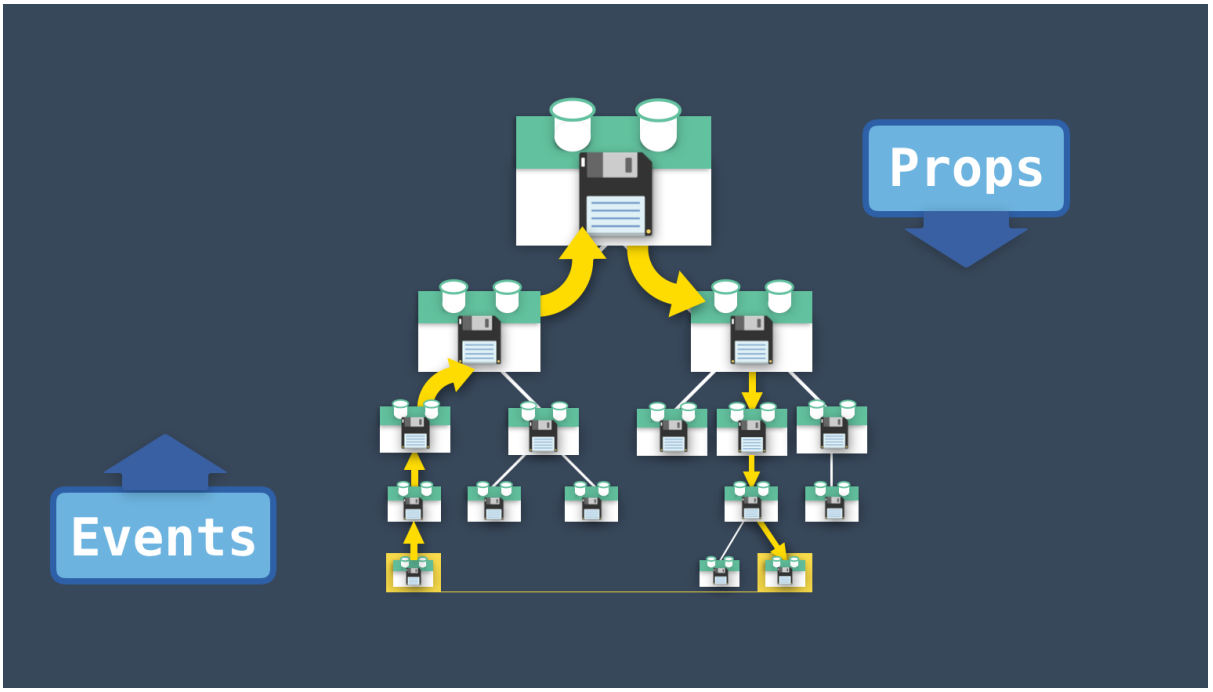
Lab8 - Vuex - state management

In Vue, as well as other front end frameworks like React, a store is a centralized location to keep up with data that is available across all the application components.

Vuex is a Vue implementation of the Flux state management pattern. It's an official state management library for Vue applications for working with data. Vuex enables developers to make complex data management easier and more efficient by using a global data store that can be accessed from all components of the Vue application for getting or setting data.

Vue Components can use different ways to communicate data between each other, such as:

- props: Props are used to pass state from a parent component to its children,
- events: Events are used to change the state of a component from its children.



Props and events can be enough for simple scenarios but once the data requirements for your application becomes complex, you'll need to implement other advanced strategies or patterns.

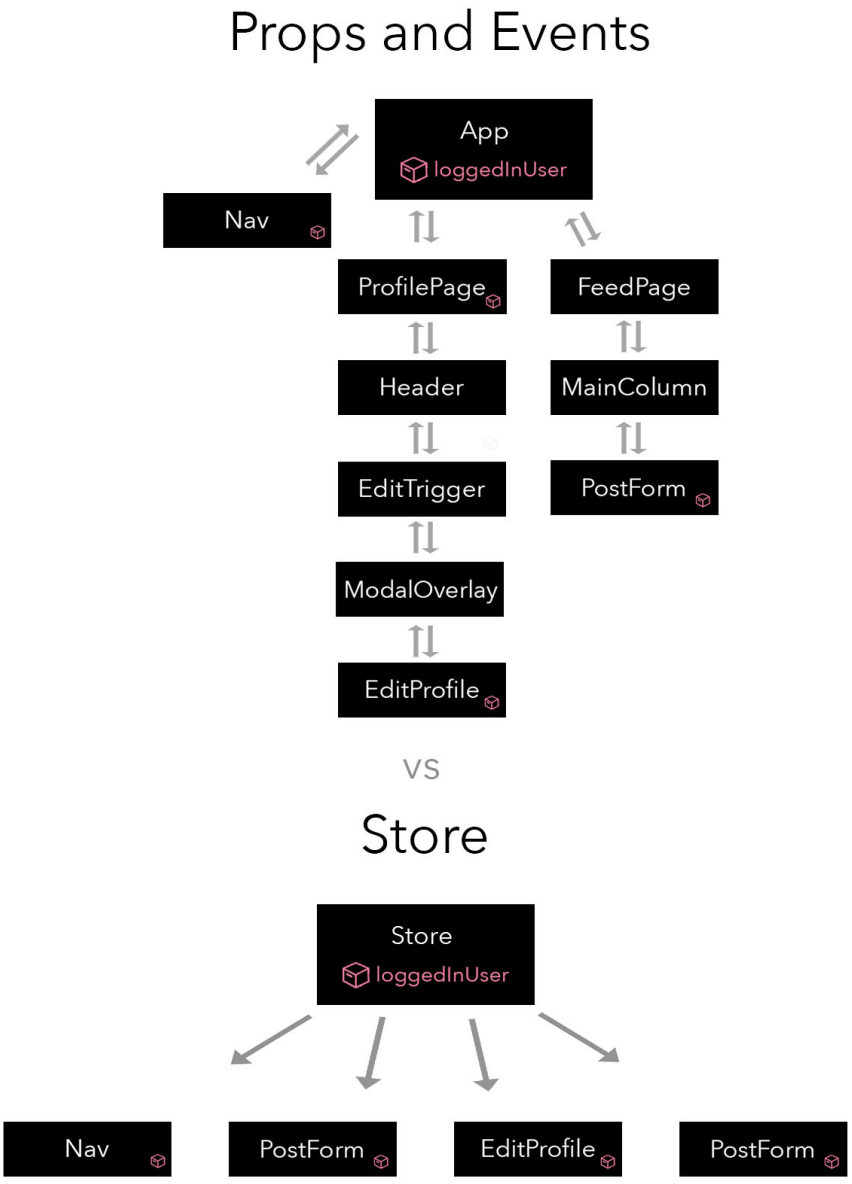
Among these patterns is the Flux pattern which aims to centralize the state across an application. In a Vue application, you can implement this pattern using Vuex.

A logged in user is a perfect example of data that belongs in a store. You'll probably need to access it in a navigation bar component to show the user at a glance that they're logged in. You'll probably need a user profile page component. You'll also probably need another component for editing that user's details. Perhaps you'll need access to that user's data in some component that makes posts in your application whether it's a forum, a blog, or a social platform, so that you can save who created the post.

If we didn't have a centralized store, we would have to pass that logged in user data down through props. If the component tree is multiple levels deep, we'd need to catch and pass the data for each level. This can quickly become unmaintainable, plus we'd also have to do the same for events if we wanted to communicate with the parents.

By having a centralized store, we reduce the complexity of needing to pass/catch props and events up and down multiple levels.

Sharing of data in this manner solves the problem of prop drilling (passing data through components that do not need it in order to get them to where they are needed).



Over time, this “family tree” of components may get quite large, creating a mental mess trying to maintain and track the state throughout your growing application.

Instead of each of our components having its own local state, we can consolidate all of our state into one place. One global location that contains the current state of our entire application. One single source of truth.

Another advantage of the store defining all the rules on how to manipulate its data, is that you can look in one place (usually in a single file) and see all the manipulation possibilities at once. You don't have to scour your entire codebase to see how different pieces of data are subject to change, you just look at the defined actions.

Installation and Setup

In order to get started with Vuex, you can install it with npm or yarn.

Firstly, create a new project with default Vue 3 settings.
Then run:

npm install vuex@4 --save

In the ‘src’ folder create another folder ‘store’ and file index.js inside. Then instantiate store via a createStore() function much like Vue 3's createApp() function.

```
// store/index.js
import { createStore } from 'vuex'
export default createStore()
```

Lastly, you register it with Vue like any other Vue plugin with the use() method.

```
// main.js
import { createApp } from 'vue'
import App from './App.vue'
import store from '@/store' // short for @/store/index

createApp(App).use(store).mount('#app')
```

Store definition

Store in Vuex is defined via an object passed to the createStore function. The object can have any of the following properties: state, getters, mutations, and actions.

```
// store/index.js
import { createStore } from 'vuex'

export default createStore({
  state:{},
  getters:{},
  mutations: {},
  actions:{}
})
```

State definition

The store's state is defined in the state property. State is just a fancy word meaning the data you want to keep in your store. You can think of it as the data property on a component but available to any of your components. You can put any kind of data you want in here like the logged in user we mentioned at the beginning.

```
  state: {
    user: { name: 'John Doe', email: 'fake@email.com', username: 'jd123'},
    posts: [],
    someString: 'this is a string',
    booleanProperty: false,
  },
```

In order to access the store's state in any component template you can use $store.state.propertyNameHere. For example, in order to access the user's name in a HelloWorld component we do the following:

```
<template>
  <h1>User name value is: {{$store.state.user.name}}</h1>
</template>
```

Or we can clean up the template a bit by using a computed property.

```
<template>
  <h1>User name value is: {{ name }}</h1>
</template>

<script>
export default {
  name: 'HelloWorld',
  computed: {
    name() { return this.$store.state.user.name }
  }
}
</script>
```

Getters

Besides the data itself stored in the state, the Vuex store can also have what's known as "getters". You can think of getters as the store version of computed properties and just like computed properties they are cached based on dependencies. All getters in Vuex are functions defined under the "getters" property and receive the state as well as all the other getters as arguments. Then whatever is returned from the function is the value of that getter.

```
// store/index.js
import { createStore } from "vuex";

export default createStore({
  state: {
    posts: ["post 1", "post 2", "post 3", "post 4"],
  },
  // the result from all the postsCount getters below is exactly the same
  // personal preference dicates how you'd like to write them
  getters: {
    // arrow function
    postsCount: (state) => state.posts.length,
    // traditional function
    postsCount: function (state) {
      return state.posts.length;
    },
    // method shorthand
    postsCount(state) {
      return state.posts.length;
    },
    // can access other getters
    postsCountMessage: (state, getters) =>
      `${getters.postsCount} posts available`,
  },
});
```

Accessing the store's getters is much the same as accessing the state except you look under the getters property instead of the state property.

```
<template>
  <p>{{$store.getters.postsCount}} posts available</p>
</template>
```

Mutations and Actions

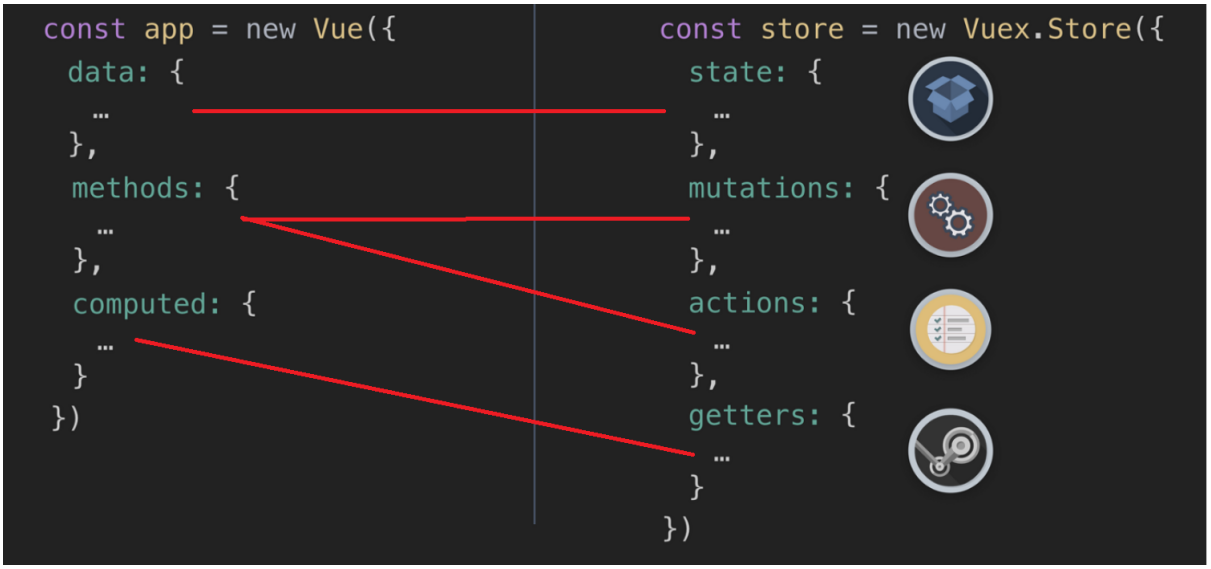
If you remember from the previous article in the series, a defining principle of a store is having rules on how the store's data can be changed. The Vuex state management pattern assigns this responsibility to actions and mutations.

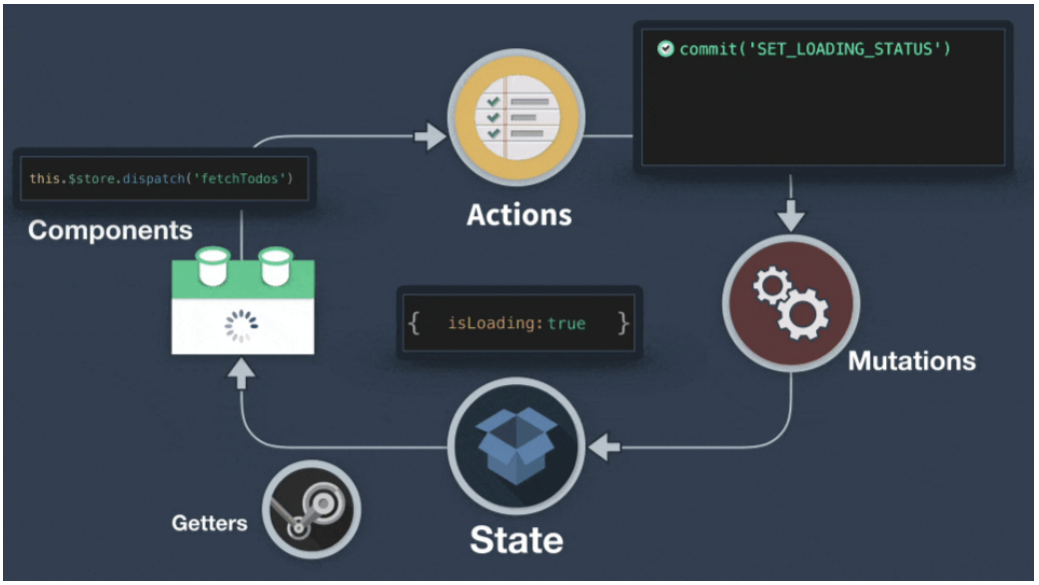
- Mutations are always synchronous and are the only thing that are allowed to change the state directly. A mutation is responsible for changing only a single piece of data.
- Actions can be synchronous or asynchronous and shouldn't change the state directly but call the mutations. Actions can invoke multiple mutations.

Also here's some more info and best practices for each.

- Mutations

    - It's convention to uppercase mutation names
    - called via **commit('MUTATION_NAME', payload)**
    - payload is optional
    - Should not contain any logic of whether to mutate the data or not
    - It's best practice to only call them from your actions (even though you do have access to calling them in components)
- Actions
    - Can be called from within other actions in the store
    - Are the primary way to change state from within components
    - called via **dispatch('actionName', payload)**
    - payload is optional
    - Can contain logic of what to mutate or not mutate





Here's an example of defining the actions and mutations:

```
export default createStore({
  state: {
    posts: ["post 1", "post 2", "post 3", "post 4"],
    user: { postsCount: 2 },
    errors: [],
  },
  mutations: {
    // convention to uppercase mutation names
    INSERT_POST(state, post) {
      state.posts.push(post);
    },
    INSERT_ERROR(state, error) {
      state.errors.push(error);
    },
    INCREMENT_USER_POSTS_COUNT(state) {
      state.user.postsCount++;
    },
  },
  actions: {
    async insertPost(context, payload) {
      //make some kind of ajax request
      try {
        await doAjaxRequest(payload);

        // can commit multiple mutations in an action
        context.commit("INSERT_POST", payload);
        context.commit("INCREMENT_USER_POSTS_COUNT");
      } catch (error) {
        context.commit("INSERT_ERROR", error);
      }
    },
  },
});
```

The context object passed to actions can be destructured, which simplifies the code:

```
 actions: {
   async insertPost({commit}, payload) {
     //make some kind of ajax request
     try {
       await doAjaxRequest(payload);

       // can commit multiple mutations in an action
       commit("INSERT_POST", payload);
       commit("INCREMENT_USER_POSTS_COUNT");
     } catch (error) {
       commit("INSERT_ERROR", error);
     }
   },
 },
```

Finally, to run actions from a component you call the dispatch method on the store and pass it the name of the action you want to run as the first argument and the payload as the second argument.

```
<template>
  <input type="text" v-model="post" />
  <button @click="$store.dispatch('insertPost', post)">Save</button>
</template>
```

It's also common to dispatch your actions from a component's method.

```
<template>
  <input type="text" v-model="post" />
  <button @click="savePost">Save</button>
</template>
<script>
export default {
  name: "HelloWorld",
  data() {
    return {
      post: "",
    };
  },
  methods: {
    savePost() {
      this.$store.dispatch("insertPost", this.post);
    },
  },
};
</script>
```

Mapping state, getters, actions, and mutations

Making the computed properties access your vuex state can get more and more verbose as you continue to access the store's state from your components. In order to lighten things up, you can use one of the Vuex helper functions mapState, passing it an array of the top level properties from the state we want access to in the component.

```
<template>
  <h1>Hello, my name is {{user.name}}</h1>
</template>
<script>
import { mapState } from 'vuex'

export default{
  computed: {
    ...mapState(['user'])
  }
}
</script>
```

The same can be done with getters using mapGetters:

```
<template>
  <p>{{postsCount}} posts available</p>
</template>
<script>
import { mapGetters } from 'vuex'
export default{
  computed:{
    ...mapGetters(['postsCount'])
  }
}
</script>
```

Of course, you have to have a getter named postsCount for this to work.

And like state and getters you can use a helper function to map actions to component methods:

```
<template>
  <input type="text" v-model="post" />
  <button @click="insertPost(post)">Save</button>
</template>
<script>
import {mapActions} from 'vuex'
export default{
  methods:{
    ...mapActions(['insertPost'])
  }
}
</script>
```

Exercise 1. Refactor your fakestore homework from the previous lesson using vuex. Switching views shouldn’t reset products’ reviews anymore. Fetch the list of products only once, when the app is created. Before assigning the list of products to the store, map all product objects and add ‘reviews: []’ property to them so that Vue can detect changes in this array. In the single product view, don’t fetch any data, just get the proper product item from the store using product id. Reloading page still should reset our reviews (as reloading reset our store state), but switching views between list of products and single product shouldn’t do this anymore.

Hint: When you will be searching in your products array for a proper product object to display in the single product view, remember that all route params are passed as string and product.id is an integer.

Organizing in Modules

As your applications grow, you'll find a single global store file becoming unbearably long and difficult to navigate. Vuex's solution to this is called modules which allow you to divide your store up into separate files based on the domain logic (such as a file for posts, another for users, another for products, and so on) and then access the state, getters, etc from the module under a particular namespace.

To create a new module, we will create a new folder ‘modules’ in the ‘store’ folder. Than, inside, create new file named eg. cart.js (following example contains cart logic):

```
export default {
  namespaced: true,

  state: {
    cart: ["bread", "rice", "beans", "turkey"],
  },

  getters: {
    // Fetch the total number of items in the cart
    totalNumberOfCartItems: (state) => {
      return state.cart.length;
    },
  },

  mutations: {
    // Add item to cart
    addItemToCart(state, payload) {
      state.cart.push(payload);
    },
    // Clear items in the cart
    emptyCart(state) {
      state.cart = [];
    },
  },

  actions: {
    test() {
      console.log('successfully run action in cart module')
    },

    checkout({ commit }, requestObject) {
      // API Call to submit the items in the cart using axios library
      fetch("submit", {
        method: "POST",
        body: requestObject
      })
        .then((response) => {
          console.log(response);
          commit("emptyCart");
        })
        .catch((error) => {
          // log error
          console.log(error);
        });
    },
  },
};
```

```
// store/index.js
import cart from "./modules/cart";
import { createStore } from "vuex";

export default createStore({
  modules: {
    cart,
  },
});
```

By default, actions and mutations are still registered under the global namespace - this allows multiple modules to react to the same action/mutation type. Getters are also registered in the global namespace by default. However, this currently has no functional purpose (it's as is to avoid breaking changes). You must be careful not to define two getters with the same name in different, non-namespaced modules, resulting in an error.

If you want your modules to be more self-contained or reusable, you can mark it as namespaced with **namespaced: true**. When the module is registered, all of its getters, actions and mutations will be automatically namespaced based on the path the module is registered at. This is something we want, so you can find namespaced: true in our cart module.

To run action located in module:

```
<template>
  <button @click="runActionInModule">Save</button>
</template>
<script>
export default{
  methods:{
    runActionInModule() {
      this.$store.dispatch('cart/test')
    }
  }
}
</script>
```

To access state in module: (first cart is module name and the second is our cart with items)

```
<template>
  <p>cart: {{ $store.state.cart.cart }}</p>
</template>
```

To access getter:

```
<template>
  <p>cart length: {{ moduleGetter }}</p>
</template>
<script>
export default {
  computed: {
    moduleGetter() { return this.$store.getters['cart/totalNumberOfCartItems'] }
  }
}
</script>
```

If you want to map your state / getters / action in your component, in the first argument give your helper name of your module. Mapping using our cart module would look like this:

```
<template>
  <p>cart: {{ cart }}</p>
  <p>cart length: {{ totalNumberOfCartItems }}</p>
  <button @click="test">run action</button>
</template>
<script>
import {mapActions, mapState, mapGetters} from 'vuex'

export default {
computed: {
  ...mapState('cart', {
    cart: state => state.cart, // -> this.cart
  }),
  ...mapGetters('cart', [
    'totalNumberOfCartItems', // -> this.totalNumberOfCartItems
  ])
},
methods: {
  ...mapActions('cart', [
    'test', // -> this.test()
  ])
}
}
</script>
```

Exercise 2. Extend your fakestore app. Move everything from the global store to the ‘products’ module. Add a ‘user’ module. Create a fake login form (accept all logins and passwords), store the login of the user in the vuex module. From now, you can’t access products list or single product view without being logged in. Unlogged users should be redirected to the login view. Delete name field from add review form, automatically get the login of logged in user when they add a review. Display the login of the user at the top of the page. Add a ‘logout’ button next to it. It should

be possible to login as user ‘x’, write some reviews, login as user ‘y’ and be able to see x's reviews and add new reviews as ‘y’ user.

Exercise 3. Add possibility to add products to ‘favourites’ list. In the ‘users’ module add an object ‘allUsersFavouriteProducts’. Its properties are users’ logins. When a new user logs in, add new property with empty array value to this object. In this array we store favourite products’ ids. Display favourite products above all products list.

```
allUsersFavouriteProducts: {

  firstUserLogin: [2, 5, 9],

  secondUserLogin: [1],

  thirdUser: []

}
```

Two way data binding

When storing form data in Vuex, it is sometimes necessary to update the value stored. The store should never be mutated directly, and an action commiting mutation should be used instead. To use v-model in our code, we need to create computed properties with getter and setter:

Our store:

```
// store/index.js
import { createStore } from "vuex";

export default createStore({
  state: {
    message: '',
  },
  actions: {
    updateMessage({commit}, newMessageValue) {
      commit('SET_MESSAGE', newMessageValue);
    }
  },
  mutations: {
    SET_MESSAGE(state, newMessageValue) {
      state.message = newMessageValue;
    }
  }
});
```

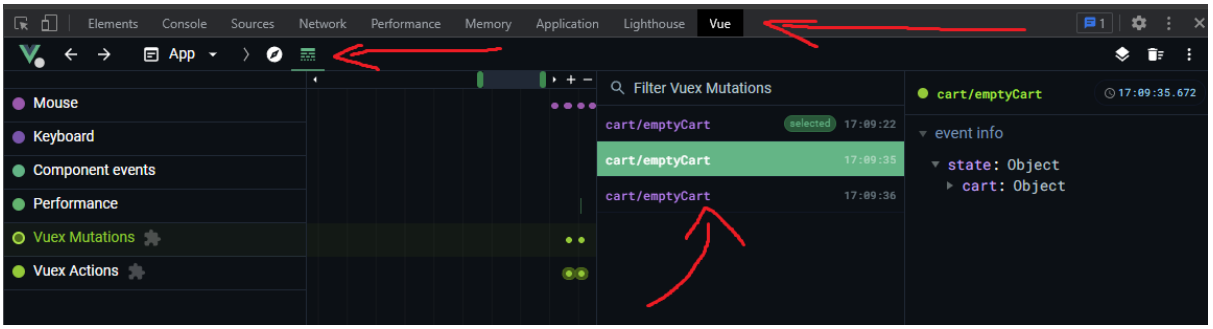
Component code:

```
<template>
  <input type="text" v-model="message">
  <p>our message: {{ message }}</p>
</template>
```

```
<script>
export default {
  computed: {
    message: {
        get () {
          return this.$store.state.message
        },
        set (value) {
          this.$store.dispatch('updateMessage', value)
        }
    }
  }
}
</script>
```

Vue devtools

You have access to all your mutations in vue devtools:



Homework:

Create a shopping list app. At the top there should be input using which you add the product to the list. Every product can be marked as bought (checkbox), edited and deleted. Clicking ‘edit’ changes the name of the product to input with its current name. Next to the input there should be a ‘confirm’ button which changes the name of the product. ‘Delete’ deletes the product. Marking as bought does not delete products from the list. Above the list there should be buttons ‘mark all as bought’ and ‘delete all marked as bought’ which do what they say. Next to it is a counter displaying how many products there are left to be bought. Below the list there should be 3 options:

All Not bought Bought

‘All’ is selected by default. Choosing ‘Not bought’ displays only not bought products, and ‘bought’ displays only already bought products. You should use store getters to filter results. **Don’t pass any data between components via props / events**, use vuex instead. The only exception is using props to pass data to Product.vue from ProductList.vue.

Upload a .zip with your homework without node_modules

Components structure:

(product in the middle with ‘submit’ is in the editing state)