Vuex

Vuex is the official state management library for Vue.js.

Its job is to share data across the components of your application.

Components in Vue.js out of the box can communicate using

- props, to pass state down to child components from a parent
- events, to change the state of a parent component from a child, or using the root component as an event bus

Sometimes things get more complex than what these simple options allow.

In this case, a good option is to centralize the state in a single store. This is what Vuex does.

Why should you use Vuex?

Vuex is not the only state management options you can use in Vue (you can use Redux too), but its main advantage is that it's official, and its integration with Vue.js is what makes it shine.

Vuex borrowed many of its ideas from the React ecosystem, as this is the Flux pattern popularized by Redux.

Components in a Vue application can have their own state. For example, an input box will store the data entered into it locally. This is perfectly fine, and components can have local state even when using Vuex.

You know that you need something like Vuex when you start doing a lot of work to pass a piece of state around.

In this case, Vuex provides a central repository store for the state, and you mutate the state by asking the store to do that.

Every component that depends on a particular piece of the state will access it using a getter on the store, which makes sure it's updated as soon as that thing changes.

Using Vuex will introduce some complexity into the application, as things need to be set up in a certain way to work correctly. But if this helps solve the unorganized props passing and event system that might grow into a spaghetti mess if too complicated, then it's a good choice.

Create the Vuex store

Now we are ready to create our Vuex store.

This file can be put anywhere. It's generally suggested to put it in the src/store/store.js file, so we'll do that.

In this file we initialize Vuex and tell Vue to use it:

```
import Vue from 'vue'

import Vuex from 'vuex'
Vue.use(Vuex)
export const store = new Vuex.Store({})
```

We export a Vuex store object, which we create using the Vuex.Store() API.

Defining a State Property

Open up our new Vue project in your code editor and visit the **/src/store.js** file. This is a file that's generated by the Vue CLI when we selected the Vuex option. This here is essentially Vuex. It's where we will define our data, mutations, actions, getters and all of that fancy stuff.

For now, let's take a look at the State section. Like I mentioned earlier, the state is where your data is defined. So, let's define a property inside of state which will allow us to set a title for our app:

```
 state: {

   title: 'My Custom Title'

},
```

Simple enough. We can add other stuff like arrays and objects, which we'll do later. Save this file and visit the **/src/components/HelloWorld.vue**. Adjust the template section as follows:

```
<template>
  <div class="hello">
    <div class="left">
      <h1>{{ title }}</h1>
    </div>
    <div class="right">

    </div>
  </div>
</template>
```

Next, in the *script* section, adjust it to match the following:

```
<script>
import { mapState } from 'vuex'

export default {
  name: 'HelloWorld',
  computed: mapState([
    'title'
  ])
}
</script>
```

First we're importing **mapState**, which is a helper that allows us to access the state from Vuex. This will give us access to our *title* property we created earlier.

Then, we're adding **computed** and calling *mapState* and passing in an array with the name of the state property we defined.

This retrieves the value of *title* and also allows us to reference it by the same name *title* through interpolation in the template section. If you save the file, you will see it now displays the title.

You can also pass in an object instead of an array with *mapState*, which allows you to do a few other things:

```
// Template Adjustment:
  <h1>{{ custom }}</h1>

// Logic:
  computed: mapState({
    custom: 'title'
})
```

Here we've defined a different alias called *custom*.

Chances are, you're going to have more than just *mapState()* as a computed property. In this case, you need to use what's called a *Object Spread Operator*. You do this adding three periods(spread operator) before *mapState* and using it as you normally would.
Make the following adjustment:

```
export default {
  name: 'HelloWorld',
  computed: {
    ...mapState([
      'title'
    ]),
    // Other properties
  }
}
```

Let's go back and define another state property as an array in **/src/store.js**:

```
state: {
    title: 'My Custom Title',
    links: [
      'http://google.com',
      'http://coursetro.com',
      'http://youtube.com'
```

```
        ]
},
```

And just in the same way, let's access it through *mapState* and render this list of links in the template:

```
<div class="left">
    <h1>{{ title }}</h1>
    <ul>
      <li v-for="(link, index) in links" v-bind:key="index">
        {{ link }}
      </li>
    </ul>
</div>
```

And in the script section:

```
...mapState([
  'title',
  'links'
]),
```

Vuex Getters

To this point, we've been using the *mapState* helper to directly access the data contained in the state. This is fine when no computations need to be made on the data itself, but when you're dealing with multiple components, you may find yourself duplicating code unnecessarily when transformations need to be made on the data. This is where Getters come in handy. Let's say for instance we want to count the number of links in our array. We can use a custom Getter for this.
Visit the */src/store.js* and add the following:

```
getters: {
  countLinks: state => {
    return state.links.length
  }
},
mutations: {},
actions: {}
```

Here, we're creating a getter called countLinks. Then, we're passing in access to the state, and returning the .length (counting the number of links in the array) on the links array contained in the state. You could do any number of things here, which could obviously be much more complex.

Going back to the original /src/components/HelloWorld.vue file, we could easily access this new getter from within this component, but I want to create another component from which we can access this getter simply for demonstration purposes. The whole point of Vuex is to allow multiple components to access the same state management, so I felt it would be necessary to create another component for this vuex tutorial.

With that said, let's make the following adjustments to our current .vue file:

```
<template>
  <div class="hello">
    <div class="left">
      <!-- Other code removed for brevity -->
    </div>
    <div class="right">
      <stats />  <!-- Add this -->
    </div>
  </div>
</template>

<script>
import Stats from '@/components/Stats.vue'  // Add this
import { mapState } from 'vuex'

export default {
  name: 'HelloWorld',
  components: {                    // Add this
    Stats
  },
  computed: {
    // Code removed for brevity
  }
}
</script>
```

Next, copy the current *HelloWorld.vue* file and rename it to *Stats.vue*. Inside of it, make the following adjustments:

```
<template>
  <div class="stats">
    <h1>A different component</h1>
    <p>There are currently {{ countLinks }} links</p>
  </div>
</template>

<script>
import { mapGetters } from 'vuex'

export default {
  name: 'Stats',
  computed: {
    ...mapGetters([
      'countLinks'
    ]),
  }
}
</script>
```

This time, instead of importing *mapState*, we're importing *mapGetters* which will give us access to our custom getters.

Then we're using it just the same as *mapState*! We call *mapGetters()* and pass in the name of the getter we want to access and displaying it via interpolation in the template

Vuex Mutations

To this point, we've only added and displayed data from the state. What about when we want to *change* the state data? That's where mutations come into play.
In our project, let's say we want to add a new link with a form input field.
We have to first create a mutation in the */src/store.js* file:

```
mutations: {
    ADD_LINK: (state, link) => {
      state.links.push(link)
    }
 },
```

We're naming this mutation *ADD_LINK* and passing in the state, and a payload named *link*. *link* will be the user-submitted value.

Then, we're simply pushing the link value to the *links* array defined in the state.
In our **/src/HelloWorld.vue** file, make the following adjustments to the template:

```
<h1>...</h1>

<!-- Add the form here -->

<form @submit.prevent="addLink">
  <input class="link-input" type="text" placeholder="Add a Link"
v-model="newLink" />
</form>

<ul>...</ul>
```

Nothing fancy is happening here. We're just creating a form that will call a method **addLink** and binding the input class to a property called *newLink*.
Let's focus on the script section now:

```
<script>
import Stats from '@/components/Stats.vue'
import { mapState, mapMutations } from 'vuex'  // Add mapMutations

export default {
 name: 'HelloWorld',
 data() {                          // Add this:
  return {
   newLink: ''
  }
 },
 components: {}, // Removed for brevity
 computed: {}, // Removed for brevity
 methods: {                          // Add this:
  ...mapMutations([
   'ADD_LINK'
  ]),
  addLink: function() {
   this.ADD_LINK(this.newLink)
   this.newLink = ''
  }
 }
}
</script>
```

Okay, the part we want to focus on is inside **methods: { }**. We're using the *mapMutations* helper to import our *ADD_LINK* mutation, and then in the *addLink()* method, we call it and pass in *this.newLink*.

Vuex Actions

Calling mutations directly in the component is for synchronous events. Should you need asyncronous functionality, you use Actions.
This time, we'll use an Action to call upon a mutation that will remove a link.
In **/src/store.js** add the following mutation and action:

```
mutations: {
   ADD_LINK: (state, link) => {
     state.links.push(link)
   },
   REMOVE_LINK: (state, link) => {       // Add this:
     state.links.splice(link, 1)
   }
 },
 actions: {
   removeLink: (context, link) => {      // Add this:
     context.commit("REMOVE_LINK", link)
   }
 }
```

So, nothing special is happening with *REMOVE_LINK*, but in *actions*, we're creating an action named *removeLink* in which the context (*context simply provides you with the same methods and properties on the store instance),* and the payload (link index) is passed.
Then we call **context.commit** where we call upon the *REMOVE_LINK* mutation and pass to it the link index. This seems redundant, but it's necessary for asynchronous operations.
Next, visit HelloWorld.vue and add the following to the template:

```
<ul>
  <li v-for="(link, index) in links" v-bind:key="index">
    {{ link }}
    <button v-on:click="removeLinks(index)" class="rm">Remove</button>  <!--
Add this -->
  </li>
</ul>
```

Then, we're going to include mapActions:

```
import { mapState, mapMutations, mapActions } from 'vuex'
```

And adjust our *methods* section:

```
methods: {
  ...mapMutations([
   'ADD_LINK'
  ]),
  ...mapActions([                // Add this
   'removeLink'
  ]),
  addLink: function() {
   this.ADD_LINK(this.newLink)
   this.newLink = '';
  },
  removeLinks: function(link) {    // Add this
   this.removeLink(link)
  }
 }
}
```

This isn't that exciting though, let's try actually demonstrating the value of using Actions to perform asynchronous operations.
Going back to */src/store.js*, let's create a new mutation and an action for removing all links with a click of a button:

```
mutations: {
  // Others removed for brevity,
  REMOVE_ALL: (state) => {                  // Add this
   state.links = []
  }
 },
 actions: {
  removeLink: (context, link) => {
   context.commit("REMOVE_LINK", link)
  },
  removeAll ({commit}) {                    // Add this
```

```
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        commit('REMOVE_ALL')
        resolve()
      }, 1500)
    })
  }
}
```

We have a new *REMOVE_ALL* mutation that empties out the array.

Then, we have a new *removeAll* action, and this time we're using *argument destructuring* to pass in commit, which makes code more simple.

We're creating a promise and simulating an operation that could take time, and calling *REMOVE_ALL* after 1.5 seconds.

We'll perform this operation in our **/src/components/Stats.vue** file:

```
<template>
  <div class="stats">
    <h1>A different component</h1>
    <p>There are currently {{ countLinks }} links</p>

    <button v-on:click="removeAllLinks">Remove all links</button>
    <p>{{ msg }}</p>
  </div>
</template>
```

In the script section:

```
<script>
import { mapGetters, mapMutations, mapActions } from 'vuex'

export default {
  name: 'Stats',
  data() {
    return {
      msg: ''
    }
  },
  computed: {
    ...mapGetters([
```

```
    'countLinks'
  ]),
 },
 methods: {
  ...mapMutations(['REMOVE_ALL']),
  ...mapActions(['removeAll']),
  removeAllLinks() {
   this.removeAll().then(() => {
    this.msg = 'They have been removed'
   });
  }
 }
}
</script>
```

Using constants for getters/mutations/actions names

If you are like me, you might have been looking with certain dislike at all the mapper helpers we have used, which relied on magic strings that should match the name of the real state, getters, mutations and actions in the store.

However, we can declare all the names in separate constants file:

```
export const mutations = {
 SET_TODOS: 'setTodos'
 ...
}
export const actions = {
 LOAD_TODOS: 'loadTodos'
 ...
}
export const getters = {
 ...
}
```

Then use ES6 features when declaring the store properties:

```
import {mutations, actions} from './constants';
export default new Vuex.Store({
 ...
 mutations: {
  [mutations.SET_TODOS](state, todos){
```

```
    state.todos = todos;
   },
   ...
 },
 actions: {
  [actions.LOAD_TODOS]({commit}){
   return axios.get('https://jsonplaceholder.typicode.com/todos')
    .then(response => {
     // get just the first 10 and commit them to the store
     const todos = response.data.slice(0, 10);
     commit(mutations.SET_TODOS, todos);
    });
  },
 },
});
```

Notice how the constants are imported and how ES6 allows us to define the actions/mutations/getters using the constant names as in [actions.LOAD_TODOS]({commit}){ rather than the standard loadTodos({commit}){ syntax. Also notice how when the action commits the mutation, it uses the constant.

Additionally, we can use the same constants when using the mapping helpers in your components:

```
 ...
 import { actions } from '@/constants';

 export default {
  name: 'todos',
  ...
  methods: {
   ...mapActions([
    actions.LOAD_TODOS,
   ]),
  },
 };
```

With this simple trick, you can avoid hardcoded strings across your components.

Vuex Modules

Another thing that will strike you is that defining the entire store and all its properties, getters, mutations and actions within a single file is not practical and won't scale well.

You could easily split them into multiple files which will be imported and combined into one store within a root store.js file. However, this would only solve the single long file problem:

- You might still find it hard to avoid clashes with state/action/mutation names.
- From any given component, it won't be easy to find the code for an action/mutation within your multiple files unless you follow naming convention.

To help you with these issues, Vuex provides the notion of modules. A **module** is basically its own store, with state, getters, mutations and actions, which is merged into a parent store.

```
// Inside /modules/todos.js
export default {
  state: {
    todos: []
  },
  getters: { ... },
  mutations: { ... },
  actions: { ... }
}

// Inside the root store.js
import todosModule from './modules/todos';
import anotherModule from './modules/another';
export default new Vuex.Store({
  modules: {
    todos: todosModule,
    another: anotherModule
  }
});
```

In its simplest usage, they are just a formal way of splitting your store code in multiple files.

NameSpacing:

However, they can also opt-in for namespacing which helps to solve the naming issues. If a module declares namespaced: true as part of its declaration, the names for state properties, getters, actions and mutations will be all namespaced to the module name.

When a namespaced module is imported by the root store as in todos: todosModule, all the resulting state, getter, action and mutation names will be prefixed with the given module name, in this case todos/. That means instead of the loadTodos action, you will have the todos/loadTodos action and so on.

Since namespacing is a common usage of Vuex, all the mapping helpers support a first argument with the name of the module you want to map from:

```
// Inside /modules/todos.js
export default {
  namespaced: true,
  state: {
    todos: []
  },
  getters: { ... },
  mutations: { ... },
  actions: { ... }
}

// Inside a component
computed: {
  ...mapState('todos', [
    'todos',
  ]),
},
methods: {
  ...mapActions('todos', [
    'loadTodos',
  ]),
},
```

Notice how the mapping helpers have a first parameter with the name of the module. Plain modules and namespaced modules allow you to scale your Vuex store as your application gets developed and more functional areas are written.