State

Single State Tree

Vuex uses a **single state tree** - that is, this single object contains all your application level state and serves as the "single source of truth". This also means usually you will have only one store for each application. A single state tree makes it straightforward to locate a specific piece of state, and allows us to easily take snapshots of the current app state for debugging purposes.

The single state tree does not conflict with modularity - in later chapters we will discuss how to split your state and mutations into sub modules.

Getting Vuex State into Vue Components

So how do we display state inside the store in our Vue components? Since Vuex stores are reactive, the simplest way to "retrieve" state from it is simply returning some store state from within a **computed property**:

```js
// let's create a Counter component
const Counter = {
  template: `<div>{{ count }}</div>`,
  computed: {
    count () {
      return store.state.count
    }
  }
}
```

Whenever `store.state.count` changes, it will cause the computed property to re-evaluate, and trigger associated DOM updates.

However, this pattern causes the component to rely on the global store singleton. When using a module system, it requires importing the store in every component that uses store state, and also requires mocking when testing the component.

Vuex provides a mechanism to "inject" the store into all child components from the root component with the `store` option (enabled by `Vue.use(Vuex)`):

```
const app = new Vue({                                                    JS
  el: '#app',
  // provide the store using the "store" option.
  // this will inject the store instance to all child components.
  store,
  components: { Counter },
  template: `
    <div class="app">
      <counter></counter>
    </div>
  `
})
```

By providing the `store` option to the root instance, the store will be injected into all child components of the root and will be available on them as `this.$store`. Let's update our `Counter` implementation:

```
const Counter = {                                                        JS
  template: `<div>{{ count }}</div>`,
  computed: {
    count () {
      return this.$store.state.count
    }
  }
}
```

The `mapState` Helper

When a component needs to make use of multiple store state properties or getters, declaring all these computed properties can get repetitive and verbose. To deal with this we can make use of the `mapState` helper which generates computed getter functions for us, saving us some keystrokes:

```js
// in full builds helpers are exposed as Vuex.mapState
import { mapState } from 'vuex'

export default {
  // ...
  computed: mapState({
    // arrow functions can make the code very succinct!
    count: state => state.count,

    // passing the string value 'count' is same as `state => state.count`
    countAlias: 'count',

    // to access local state with `this`, a normal function must be used
    countPlusLocalState (state) {
      return state.count + this.localCount
    }
  })
}
```

We can also pass a string array to `mapState` when the name of a mapped computed property is same as a state sub tree name.

```js
computed: mapState([
  // map this.count to store.state.count
  'count'
])
```

Object Spread Operator

Note that `mapState` returns an object. How do we use it in combination with other local computed properties? Normally, we'd have to use a utility to merge multiple objects into one so that we can pass the final object to `computed`. However with the **object spread operator** (which is a stage-3 ECMAScript proposal), we can greatly simplify the syntax:

```js
computed: {
  localComputed () { /* ... */ },
  // mix this into the outer object with the object spread operator
  ...mapState({
    // ...
  })
}
```

Components Can Still Have Local State

Using Vuex doesn't mean you should put **all** the state in Vuex. Although putting more state into Vuex makes your state mutations more explicit and debuggable, sometimes it could also make the code more verbose and indirect. If a piece of state strictly belongs to a single component, it could be just fine leaving it as local state. You should weigh the trade-offs and make decisions that fit the development needs of your app.

Getters

Sometimes we may need to compute derived state based on store state, for example filtering through a list of items and counting them:

```js
computed: {
  doneTodosCount () {
    return this.$store.state.todos.filter(todo => todo.done).length
  }
}
```

If more than one component needs to make use of this, we have to either duplicate the function, or extract it into a shared helper and import it in multiple places - both are less than ideal.

Vuex allows us to define "getters" in the store. You can think of them as computed properties for stores. Like computed properties, a getter's result is cached based on its dependencies, and will only re-evaluate when some of its dependencies have changed.

Getters will receive the state as their 1st argument:

```js
const store = new Vuex.Store({
  state: {
    todos: [
      { id: 1, text: '...', done: true },
      { id: 2, text: '...', done: false }
    ]
  },
  getters: {
    doneTodos: state => {
      return state.todos.filter(todo => todo.done)
    }
  }
})
```

The getters will be exposed on the `store.getters` object:

```js
store.getters.doneTodos // -> [{ id: 1, text: '...', done: true }]
```

Getters will also receive other getters as the 2nd argument:

```js
getters: {
  // ...
  doneTodosCount: (state, getters) => {
    return getters.doneTodos.length
  }
}
store.getters.doneTodosCount // -> 1
```

We can now easily make use of it inside any component:

```
computed: {
  doneTodosCount () {
    return this.$store.getters.doneTodosCount
  }
}
```
JS

You can also pass arguments to getters by returning a function. This is particularly useful when you want to query an array in the store:

```
getters: {
  // ...
  getTodoById: (state, getters) => (id) => {
    return state.todos.find(todo => todo.id === id)
  }
}
store.getters.getTodoById(2) // -> { id: 2, text: '...', done: false }
```
JS

The `mapGetters` Helper

The `mapGetters` helper simply maps store getters to local computed properties:

```
import { mapGetters } from 'vuex'

export default {
  // ...
  computed: {
    // mix the getters into computed with object spread operator
    ...mapGetters([
      'doneTodosCount',
      'anotherGetter',
      // ...
    ])
  }
}
```
JS

If you want to map a getter to a different name, use an object:

```
...mapGetters({
  // map `this.doneCount` to `store.getters.doneTodosCount`
  doneCount: 'doneTodosCount'
})
```
JS

Mutations

The only way to actually change state in a Vuex store is by committing a mutation. Vuex mutations are very similar to events: each mutation has a string **type** and a **handler**. The handler function is where we perform actual state modifications, and it will receive the state as the first argument:

```js
const store = new Vuex.Store({
  state: {
    count: 1
  },
  mutations: {
    increment (state) {
      // mutate state
      state.count++
    }
  }
})
```

You cannot directly call a mutation handler. Think of it more like event registration: "When a mutation with type `increment` is triggered, call this handler." To invoke a mutation handler, you need to call `store.commit` with its type:

```js
store.commit('increment')
```

Commit with Payload

You can pass an additional argument to `store.commit`, which is called the **payload** for the mutation:

```js
// ...
mutations: {
  increment (state, n) {
    state.count += n
  }
}
store.commit('increment', 10)
```

In most cases, the payload should be an object so that it can contain multiple fields, and the recorded mutation will also be more descriptive:

```js
// ...
mutations: {
  increment (state, payload) {
    state.count += payload.amount
  }
}
store.commit('increment', {
  amount: 10
})
```

Object-Style Commit

An alternative way to commit a mutation is by directly using an object that has a `type` property:

```js
store.commit({
  type: 'increment',
  amount: 10
})
```

When using object-style commit, the entire object will be passed as the payload to mutation handlers, so the handler remains the same:

```js
mutations: {
  increment (state, payload) {
    state.count += payload.amount
  }
}
```

Mutations Follow Vue's Reactivity Rules

Since a Vuex store's state is made reactive by Vue, when we mutate the state, Vue components observing the state will update automatically. This also means Vuex mutations are subject to the same reactivity caveats when working with plain Vue:

1. Prefer initializing your store's initial state with all desired fields upfront.
2. When adding new properties to an Object, you should either:
   - Use `Vue.set(obj, 'newProp', 123)`, or
   - Replace that Object with a fresh one. For example, using the stage-3 **object spread syntax** we can write it like this:

     ```js
     state.obj = { ...state.obj, newProp: 123 }
     ```

Using Constants for Mutation Types

It is a commonly seen pattern to use constants for mutation types in various Flux implementations. This allows the code to take advantage of tooling like linters, and putting all constants in a single file allows your collaborators to get an at-a-glance view of what mutations are possible in the entire application:

```js
// mutation-types.js
export const SOME_MUTATION = 'SOME_MUTATION'
```

```js
// store.js
import Vuex from 'vuex'
import { SOME_MUTATION } from './mutation-types'

const store = new Vuex.Store({
  state: { ... },
  mutations: {
    // we can use the ES2015 computed property name feature
    // to use a constant as the function name
    [SOME_MUTATION] (state) {
      // mutate state
    }
  }
})
```

Whether to use constants is largely a preference - it can be helpful in large projects with many developers, but it's totally optional if you don't like them.

Mutations Must Be Synchronous

One important rule to remember is that **mutation handler functions must be synchronous**. Why? Consider the following example:

```js
mutations: {
  someMutation (state) {
    api.callAsyncMethod(() => {
      state.count++
    })
  }
}
```

Now imagine we are debugging the app and looking at the devtool's mutation logs. For every mutation logged, the devtool will need to capture a "before" and "after" snapshots of the state. However, the asynchronous callback inside the example mutation above makes that impossible: the callback is not called yet when the mutation is committed, and there's no way for the devtool to know when the callback will actually be called - any state mutation performed in the callback is essentially un-trackable!

Committing Mutations in Components

You can commit mutations in components with `this.$store.commit('xxx')`, or use the `mapMutations` helper which maps component methods to `store.commit` calls (requires root `store` injection):

```js
import { mapMutations } from 'vuex'

export default {
  // ...
  methods: {
    ...mapMutations([
      'increment', // map `this.increment()` to `this.$store.commit('increment')`

      // `mapMutations` also supports payloads:
      'incrementBy' // map `this.incrementBy(amount)` to `this.$store.commit('incrementBy', amount)`
    ]),
    ...mapMutations({
      add: 'increment' // map `this.add()` to `this.$store.commit('increment')`
    })
  }
}
```

On to Actions

Asynchronicity combined with state mutation can make your program very hard to reason about. For example, when you call two methods both with async callbacks that mutate the state, how do you know when they are called and which callback was called first? This is exactly why we want to separate the two concepts. In Vuex, **mutations are synchronous transactions**:

```js
store.commit('increment')
// any state change that the "increment" mutation may cause
// should be done at this moment.
```

To handle asynchronous operations, let's introduce **Actions**.

Actions

Actions are similar to mutations, the differences being that:

- Instead of mutating the state, actions commit mutations.
- Actions can contain arbitrary asynchronous operations.

Let's register a simple action:

```js
const store = new Vuex.Store({
  state: {
    count: 0
  },
  mutations: {
    increment (state) {
      state.count++
    }
  },
  actions: {
    increment (context) {
      context.commit('increment')
    }
  }
})
```

Action handlers receive a context object which exposes the same set of methods/properties on the store instance, so you can call `context.commit` to commit a mutation, or access the state and getters via `context.state` and `context.getters`. We will see why this context object is not the store instance itself when we introduce **Modules** later.

In practice, we often use ES2015 **argument destructuring** to simplify the code a bit (especially when we need to call `commit` multiple times):

```js
actions: {
  increment ({ commit }) {
    commit('increment')
  }
}
```

Dispatching Actions

Actions are triggered with the `store.dispatch` method:

```js
store.dispatch('increment')
```

This may look dumb at first sight: if we want to increment the count, why don't we just call `store.commit('increment')` directly? Well, remember that **mutations must be synchronous**? Actions don't. We can perform **asynchronous** operations inside an action:

```js
actions: {
  incrementAsync ({ commit }) {
    setTimeout(() => {
      commit('increment')
    }, 1000)
  }
}
```

Actions support the same payload format and object-style dispatch:

```js
// dispatch with a payload
store.dispatch('incrementAsync', {
  amount: 10
})

// dispatch with an object
store.dispatch({
  type: 'incrementAsync',
  amount: 10
})
```

A more practical example of real-world actions would be an action to checkout a shopping cart, which involves **calling an async API** and **committing multiple mutations**:

```js
actions: {
  checkout ({ commit, state }, products) {
    // save the items currently in the cart
    const savedCartItems = [...state.cart.added]
    // send out checkout request, and optimistically
    // clear the cart
    commit(types.CHECKOUT_REQUEST)
    // the shop API accepts a success callback and a failure callback
    shop.buyProducts(
      products,
      // handle success
      () => commit(types.CHECKOUT_SUCCESS),
      // handle failure
      () => commit(types.CHECKOUT_FAILURE, savedCartItems)
    )
  }
}
```

Note we are performing a flow of asynchronous operations, and recording the side effects (state mutations) of the action by committing them.

Dispatching Actions in Components

You can dispatch actions in components with `this.$store.dispatch('xxx')`, or use the `mapActions` helper which maps component methods to `store.dispatch` calls (requires root `store` injection):

```
import { mapActions } from 'vuex'

export default {
  // ...
  methods: {
    ...mapActions([
      'increment', // map `this.increment()` to `this.$store.dispatch('increment')`

      // `mapActions` also supports payloads:
      'incrementBy' // map `this.incrementBy(amount)` to `this.$store.dispatch('incrementBy', amount)
    ]),
    ...mapActions({
      add: 'increment' // map `this.add()` to `this.$store.dispatch('increment')`
    })
  }
}
```

JS

Composing Actions

Actions are often asynchronous, so how do we know when an action is done? And more importantly, how can we compose multiple actions together to handle more complex async flows?

The first thing to know is that `store.dispatch` can handle Promise returned by the triggered action handler and it also returns Promise:

```
actions: {
  actionA ({ commit }) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        commit('someMutation')
        resolve()
      }, 1000)
    })
  }
}
```

JS

Now you can do:

```
store.dispatch('actionA').then(() => {
  // ...
})
```

JS

And also in another action:

```js
actions: {
  // ...
  actionB ({ dispatch, commit }) {
    return dispatch('actionA').then(() => {
      commit('someOtherMutation')
    })
  }
}
```

Finally, if we make use of **async** / **await**, a JavaScript feature landing very soon, we can compose our actions like this:

```js
// assuming `getData()` and `getOtherData()` return Promises

actions: {
  async actionA ({ commit }) {
    commit('gotData', await getData())
  },
  async actionB ({ dispatch, commit }) {
    await dispatch('actionA') // wait for `actionA` to finish
    commit('gotOtherData', await getOtherData())
  }
}
```

> **It's possible for a `store.dispatch` to trigger multiple action handlers in different modules. In such a case the returned value will be a Promise that resolves when all triggered handlers have been resolved.**

Modules

Due to using a single state tree, all state of our application is contained inside one big object. However, as our application grows in scale, the store can get really bloated.

To help with that, Vuex allows us to divide our store into **modules**. Each module can contain its own state, mutations, actions, getters, and even nested modules - it's fractal all the way down:

```js
const moduleA = {
  state: { ... },
  mutations: { ... },
  actions: { ... },
  getters: { ... }
}

const moduleB = {
  state: { ... },
  mutations: { ... },
  actions: { ... }
}

const store = new Vuex.Store({
  modules: {
    a: moduleA,
    b: moduleB
  }
})

store.state.a // -> `moduleA`'s state
store.state.b // -> `moduleB`'s state
```

Module Local State

Inside a module's mutations and getters, the first argument received will be **the module's local state**.

```js
const moduleA = {
  state: { count: 0 },
  mutations: {
    increment (state) {
      // `state` is the local module state
      state.count++
    }
  },

  getters: {
    doubleCount (state) {
      return state.count * 2
    }
  }
}
```

Similarly, inside module actions, `context.state` will expose the local state, and root state will be exposed as `context.rootState`:

```js
const moduleA = {
  // ...
  actions: {
    incrementIfOddOnRootSum ({ state, commit, rootState }) {
      if ((state.count + rootState.count) % 2 === 1) {
        commit('increment')
      }
    }
  }
}
```

Also, inside module getters, the root state will be exposed as their 3rd argument:

```js
const moduleA = {
  // ...
  getters: {
    sumWithRootCount (state, getters, rootState) {
      return state.count + rootState.count
    }
  }
}
```

Namespacing

By default, actions, mutations and getters inside modules are still registered under the **global namespace** - this allows multiple modules to react to the same mutation/action type.

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. For example:

```js
const store = new Vuex.Store({
  modules: {
    account: {
      namespaced: true,

      // module assets
      state: { ... }, // module state is already nested and not affected by namespace option
      getters: {
        isAdmin () { ... } // -> getters['account/isAdmin']
      },
      actions: {
        login () { ... } // -> dispatch('account/login')
      },
      mutations: {
        login () { ... } // -> commit('account/login')
      },

      // nested modules
      modules: {
        // inherits the namespace from parent module
        myPage: {
          state: { ... },
          getters: {
            profile () { ... } // -> getters['account/profile']
          }
        },

        // further nest the namespace
        posts: {
          namespaced: true,

          state: { ... },
          getters: {
            popular () { ... } // -> getters['account/posts/popular']
          }
        }
      }
    }
  }
})
```

Namespaced getters and actions will receive localized `getters`, `dispatch` and `commit`. In other words, you can use the module assets without writing prefix in the same module. Toggling between namespaced or not does not affect the code inside the module.

Accessing Global Assets in Namespaced Modules

If you want to use global state and getters, `rootState` and `rootGetters` are passed as the 3rd and 4th arguments to getter functions, and also exposed as properties on the `context` object passed to action functions.

To dispatch actions or commit mutations in the global namespace, pass `{ root: true }` as the 3rd argument to `dispatch` and `commit`.

```
modules: {                                                                    JS
  foo: {
    namespaced: true,

    getters: {
      // `getters` is localized to this module's getters
      // you can use rootGetters via 4th argument of getters
      someGetter (state, getters, rootState, rootGetters) {
        getters.someOtherGetter // -> 'foo/someOtherGetter'
        rootGetters.someOtherGetter // -> 'someOtherGetter'
      },
      someOtherGetter: state => { ... }
    },

    actions: {
      // dispatch and commit are also localized for this module
      // they will accept `root` option for the root dispatch/commit
      someAction ({ dispatch, commit, getters, rootGetters }) {
        getters.someGetter // -> 'foo/someGetter'
        rootGetters.someGetter // -> 'someGetter'

        dispatch('someOtherAction') // -> 'foo/someOtherAction'
        dispatch('someOtherAction', null, { root: true }) // -> 'someOtherAction'

        commit('someMutation') // -> 'foo/someMutation'
        commit('someMutation', null, { root: true }) // -> 'someMutation'
      },
      someOtherAction (ctx, payload) { ... }
    }
  }
}
```

Binding Helpers with Namespace

When binding a namespaced module to components with the `mapState`, `mapGetters`, `mapActions` and `mapMutations` helpers, it can get a bit verbose:

```js
computed: {
  ...mapState({
    a: state => state.some.nested.module.a,
    b: state => state.some.nested.module.b
  })
},
methods: {
  ...mapActions([
    'some/nested/module/foo',
    'some/nested/module/bar'
  ])
}
```

In such cases, you can pass the module namespace string as the first argument to the helpers so that all bindings are done using that module as the context. The above can be simplified to:

```js
computed: {
  ...mapState('some/nested/module', {
    a: state => state.a,
    b: state => state.b
  })
},
methods: {
  ...mapActions('some/nested/module', [
    'foo',
    'bar'
  ])
}
```

Furthermore, you can create namespaced helpers by using `createNamespacedHelpers`. It returns an object having new component binding helpers that are bound with the given namespace value:

```js
import { createNamespacedHelpers } from 'vuex'

const { mapState, mapActions } = createNamespacedHelpers('some/nested/module')

export default {
  computed: {
    // look up in `some/nested/module`
    ...mapState({
      a: state => state.a,
      b: state => state.b
    })
  },
  methods: {
    // look up in `some/nested/module`
    ...mapActions([
      'foo',
      'bar'
    ])
  }
}
```

Caveat for Plugin Developers

You may care about unpredictable namespacing for your modules when you create a **plugin** that provides the modules and let users add them to a Vuex store. Your modules will be also namespaced if the plugin users add your modules under a namespaced module. To adapt this situation, you may need to receive a namespace value via your plugin option:

```js
// get namespace value via plugin option
// and returns Vuex plugin function
export function createPlugin (options = {}) {
  return function (store) {
    // add namespace to plugin module's types
    const namespace = options.namespace || ''
    store.dispatch(namespace + 'pluginAction')
  }
}
```

Dynamic Module Registration

You can register a module **after** the store has been created with the `store.registerModule` method:

```
// register a module `myModule`
store.registerModule('myModule', {
  // ...
})

// register a nested module `nested/myModule`
store.registerModule(['nested', 'myModule'], {
  // ...
})
```
JS

The module's state will be exposed as `store.state.myModule` and `store.state.nested.myModule`.

Dynamic module registration makes it possible for other Vue plugins to also leverage Vuex for state management by attaching a module to the application's store. For example, the **`vuex-router-sync`** library integrates vue-router with vuex by managing the application's route state in a dynamically attached module.

You can also remove a dynamically registered module with `store.unregisterModule(moduleName)`. Note you cannot remove static modules (declared at store creation) with this method.

Module Reuse

Sometimes we may need to create multiple instances of a module, for example:

- Creating multiple stores that use the same module (e.g. To **avoid stateful singletons in the SSR** when the `runInNewContext` option is `false` or `'once'`);
- Register the same module multiple times in the same store.

If we use a plain object to declare the state of the module, then that state object will be shared by reference and cause cross store/module state pollution when it's mutated.

This is actually the exact same problem with `data` inside Vue components. So the solution is also the same - use a function for declaring module state (supported in 2.3.0+):

```
const MyReusableModule = {
  state () {
    return {
      foo: 'bar'
    }
  },
  // mutations, actions, getters...
}
```
JS