Vuex getters are great, but don’t overuse them

codeburst.io/vuex-getters-are-great-but-dont-overuse-them-9c946689b414

As it states in the Bunny Theory of Code, the code tends to multiply in a quick way. And even with such great tools as Vue and Vuex it’s easy to make a mistake that might grow later all over the project. Here I will talk about Vuex and its getters, and how they might be used in a wrong way. And while dealing with that, I tried to figure out what Vuex getters are good for and how they work under the hood.

Note: It would be nice to have some basic knowledge on Vue and Vuex libraries to read further.

A bit about dummy getters

Getters are a part of Vuex store and they are used to calculate data based on store state. Basically, they are one of the many things that make Vue and Vuex shine together. As the documentation says:

> *Vuex allows us to define “getters” in the store. You can think of them as computed properties for stores.*
>
> *https://vuex.vuejs.org/en/getters.html*

Here are a few things that are great about getters:

- They are easily accessible inside both components and Vuex actions;
- They cache data and smartly update themselves when the state changes;
- They can return functions, so that it’s possible to pass additional arguments to calculate data based on them.

In fact getters are so great that it is easy to fall in the pit of overusing them extensively. And the mistake I am talking about here is to use getters in a “dummy” way, which can be illustrated in the following example. Let’s create a simple store with a few getters:

```
const store = new Vuex.Store({
  state: {
    movies: [
      { id: 1, name: 'La La Land', watched: true },
      { id: 2, name: 'Moonlight', watched: false }
    ],
    genres: [
      { id: 1, name: 'Comedy' },
      { id: 2, name: 'Drama' }
    ]
  },
  getters: {
    movies: state => state.movies,
    genres: state => state.genres
  }
})
```

These getters are then used in a component to reach the store through them:

```
computed: {
  movies() {
    return this.$store.getters.movies
  },
  genres() {
    return this.$store.getters.genres
  }
}
```

In this example we use getters in a dummy way, just to return a specific property from the store. It is an overdo, because the code can be written in a much more efficient way.

mapGetters and mapState

First thing we can do is to use `mapGetters` to bind store getters to local computed properties.

> The *`mapGetters`* helper simply maps store getters to local computed properties https://vuex.vuejs.org/en/getters.html

Now component looks simpler:

```
import { mapGetters } from 'vuex'
```

```
export default {
  computed: {
    ...mapGetters([
      'movies',
      'genres'
    ])
  }
}
```

But we still have dummy getters inside the store. The good news is that we can get rid of them completely, because there is the `mapState`:

> *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*

> *https://vuex.vuejs.org/en/state.html*

Let's get back to our example and use `mapState` instead of `mapGetters`:

```
import { mapState } from 'vuex'

export default {
  computed: {
    ...mapState(['movies', 'genres'])
  }
}
```

Here `mapState` creates simple getters and maps them into computed properties, so that they can easily be used inside components. After that we can remove getters from our store completely, because now *mapState* takes care of everything:

```
const store = new Vuex.Store({
  state: {
    movies: [
      { id: 1, name: 'La La Land', watched: true },
      { id: 2, name: 'Moonlight', watched: false }
    ],
    genres: [
      { id: 1, name: 'Comedy' },
      { id: 2, name: 'Drama' }
    ]
  }
})
```

If you have Vuex store separated into modules, you can't use the array shorthand syntax here. Instead, you need to use the object notation as shown below:

```
const movieListModule = {
  state: { ... }
}

const store = new Vuex.Store({
  modules: {
    movieList: movieListModule
  }
})
```

In this case `mapState` will look like this (as an option you can use a few nice shortcuts that Vue provides):

```
import { mapState } from 'vuex'

export default {
  computed: mapState({
    movies: state => state.movieList.movies,
    genres: state => state.movieList.genres
  })
}
```

As a result, `mapState` helps us to get rid of dummy getters and directly map state to components, so now we can use getters only when they are really needed.

Let getters shine

Let's say, that in our example we need to get an array of movies already watched. If it's only about one component, then it's as simple as putting this logic into a calculated property inside of it. But what if another component also needs the same thing?

We might consider storing calculated data inside the store, but since in our case it's a derived data based on movies list, it comes with responsibility to keep it in sync wherever the original list changes.

That is when getters might be the best solution. We can define a simple getter function to get movies based on `watched` property:

```
const store = new Vuex.Store({
  state: {
    movies: [
      { id: 1, name: 'La La Land', watched: true },
      { id: 2, name: 'Moonlight', watched: false }
    ],
    genres: [
      { id: 1, name: 'Comedy' },
      { id: 2, name: 'Drama' }
    ]
  },
  getters: {
    watchedMovies: state => {
      return state.movies.filter((movie) => movie.watched))
    }
  }
})
```

Now every component and every action can access the `watchedMovies` property and it's cached, meaning it won't be recalculated till the store data that is involved changes.

It is also possible to pass arguments to getters and calculate data based on it. It can be achieved by returning a function:

```
getters: {
  getMovieById: (state, getters) => (id) => {
    return state.movies.find(movie => movie.id === id)
  }
}
```

Getters Cache

Caching is one of the key features of getters, which actually makes them more preferable over simple helper functions. Getters cache works in a way that if a getter is calculated once, it won't be recalculated till the part of a store it depends on changes as well. The detailed explanation on how getters caching works can be found in the Appendix.

But be aware that since Vue's computed properties were designed for data storing, and not for functions, the computed properties cache for parameterised getters does not work. This way the underlying function is executed each time getter is accessed.

Let's try an example to show the difference in caching between simple and parameterised getters:

The example shows that getters are updated only when the part of store they are using is updated. And while in the basic case the result is cached, in the case of parameterised getters the result is recalculated each time the getter is called. It's a good point to keep in mind if you are using parameterised getters extensively.

Conclusion

Getters have an important role as a part of Vuex store and it's our responsibility to use them properly—to compute properties based on state and to provide them in convenient way for both components and actions, so that they won't be recalculated each time.

TL;DR

- Using dummy getters to return a part of a store is an overdo
- *mapState* is a simple and elegant solution which allows us to get rid of dummy getters, by directly mapping the store data to a component
- Use getters only for non-trivial data calculation wich is required in few components
- Getters cache is based on Vue's calculate properties
- Parameterised getters are not cached!

Thanks for reading this article. Please leave claps if you find it useful! And I will be happy to hear any feedback.

Appendix

Let's look under the hood of Vuex to see how getters caching works. First, all the getters are gathered together under the `computed` variable:

```
// bind store public getters
store.getters = {}
const wrappedGetters = store._wrappedGetters
const computed = {}

forEachValue(wrappedGetters, (fn, key) => {
  // use computed to leverage its lazy-caching mechanism
  computed[key] = () => fn(store)

Object.defineProperty(store.getters, key, {
    get: () => store._vm[key],
    enumerable: true // for local getters
  })
})
```

Then Vuex makes a smart move and simply uses the instance of Vue and it's existing reactive mechanism for getters caching. This is achieved by putting getters inside a `computed` property of a Vue instance:

```
// use a Vue instance to store the state tree
// suppress warnings just in case the user has added
// some funky global mixins
const silent = Vue.config.silent
Vue.config.silent = true
store._vm = new Vue({
  data: {
    $$state: state
  },
  computed
})
```

The reactive update of computed properties in Vue is built upon using of the Observable Data Model and getters/setters. Basically, during it's evaluation a computed property will register it's dependencies on

observable properties, and later observable properties will update computed properties, when they are changed. The basic approach of dependency tracking is described here. ^