Creating a product configurator with Vue #2

Part 1: The Vuex store

Pim Hoogheimstra · Follow
May 31, 2019 · 6min read

In this post I'll describe the setup of the Vuex store for the product configurator that was introduced in part 1. The product configurator is part of a fictive webshop where customers may configure and order personal products.

Since usability becomes more and more important these days, this blog series covers a product configurator from concept to implementation.

Photo by Helena Rose on Unsplash

Vuex

The following actions may take place during the configuration of the postcards:

- add a postcard
- remove a postcard
- update the total price
- reset a configuration for a given postcard
- add/ or cast and postcard to the checkout

These will become the actions for the Vuex store. I'll discuss the setup of this part of the store in a minute.

Vuex modules

Pim's wand ez Vuex modules. For a small project like this it might not be necessary, but your future self will be thankful if the project grows 😉

Moreover, setting up the modules is not very hard. But what modules do we have in this project?

Closely there is a configurator module consisting of all aspects related to postcards. It's simple to have more accounts in the webshop. An account, with login, logout and so on is a good candidate for another module. In addition, the cart could be a module. However, the latter is out of scope for the current post, so let's assume we only need modules for the configurator and for accounts. The actions for the configurator are described above, for the account module I come to the following set of actions:

Vuex modules implemented

Implementing the store with modules is quite easy. However, instead of the .store.js file that comes with Vuex by default, we need a /store folder holding an index.js file and a modules folder that holds /account.js and /configurator.js. Each module consists of its own state, actions, mutations, getters and (possibly) submodules of required by the project. The store looks a lot like this:

```
import Vue from 'vue'
import Vuex from 'vuex'
import configurator from './modules/configurator'
import account from './modules/account'

Vue.use(Vuex)

export default new Vuex.Store({
  modules: {
    configurator,
    account
  }
})
```

store/index.js hosted with ❤ by GitHub view raw

The store itself is imported in main.js. This happens by default when adding Vuex to the project during setup with @vue-cli:

```
import Vue from 'vue'
import App from 'App'
import configurator from './modules/configurator'
import account from './modules/account'

Vue.use(Vuex)

export default new Vuex.Store({
  modules: {
    configurator,
    account
  }
})
```

store/index.js hosted with ❤ by GitHub view raw

Configurator module

The state of the configurator module contains a products array and a price. We initialize the price at € 1.00. A single product is initialized in the products array. It's an object with a unique id and a single object holding all the properties a postcard may have:

```
// store.index.js
const state = {
  products: [
    {
      id: uuid(),
      properties: {
        template: 'A basic template',
        text: 'Hello, this is a test text. Merry',
        amount: 1,
        shape: 'portrait',
        paper: 'Glossy',
        font: 'Arial',
        image: 'https://image.jpg',
        quality: 'Normal',
        ...
      },
    },
  ],
  price: 1.00,
},

configurator/state.js hosted with ❤ by GitHub view raw
```

Note that the id field is generated using the uuid@ function. This is a truly unique string and we need it as a key later on. Otherwise the remove product feature won't work as expected!

Our actions are defined as follows:

```
// store.index.js
const actions = {
  updateProperties({ commit },
    { product, properties, index }) {
    commit('updateProperties')
  },

  resetProductPropertiesAt({ commit },
    { index }) {
    commit('resetProductPropertiesAt')
  },

  addProductAt({ commit },
    { index }) {
    commit('addProductAt')
  },

  castProductAt({ commit },
    { index }) {
    commit('castProductAt')
  },

  resetProductPropertiesAt({ commit },
    { index }) {
    commit('resetProductPropertiesAt')
  },

  updateProductPrice({ commit },
    { price }) {
    commit('updateProductPrice', price)
  },
},

configurator/actions.js hosted with ❤ by GitHub view raw
```

The current application has no backend, so everything is synchronous and actions are not required. Hence, we could have put mutations directly. However, adding actions makes it possible to add asynchronous behavior in the future. An example is a dedicated API calculate the price of a product.

The mutations are defined below:

```
// store.index.js
const mutations = {
  updateProperties(state,
    { product, properties, index }) {
    product.properties = properties
  },

  resetProductPropertiesAt(state, product) {
    const index = products.findIndexOf(product => product.id == product.id)
    products[index].properties = {}
  },

  addProductAt(state, index) {
    const product = { id: uuid(), properties: {} }
    const index = products.findIndexOf(product => product.id == product.id)
    products.splice(index, 0, product)
    products[index].properties = {}
  },

  castProductAt(state, index) {
    const index = products.findIndexOf(product => product.id == product.id)
    products.splice(index, 1)
    products[index].properties = {}
  },

  resetProductPropertiesAt(state, index) {
    const index = products.findIndexOf(product => product.id == product.id)
    products[index].properties = {}
  },

  updateProductPrice(state, price) {
    state.price = price
  },
},

configurator/mutations.js hosted with ❤ by GitHub view raw
```

The addProduct mutation simply pushes a new product object onto the products array. The resetProductProperties mutation on the other hand removes the product with a certain index. Add to cast functionality is outside the scope of this project, we could think of cleaning the products array and reset the price. The updateProductPrice mutation updates the price of a product and uses the findIndexed property using lodash's/underscore's findIndex method. The productPrice function is a very simple function:

```
const productPrice = (product) => {
  const { shape, paper, quality } = product.properties
  const price = 0.95
  const extra = 0.05

  price += 0.05 * (shape.length.toFixed) // 1.0 for shape is landscape
  price += 0.05 * (paper.toLowerCase() == 'glossy') // 1.0 if paper is glossy
  price += 0.05 * (quality == 'Extra')
  price += 0.05 * (quality == 'Normal')

  price = Math.round(price * 100) / 100
  return price
}

export { productPrice }

utilities/productPrice.js hosted with ❤ by GitHub view raw
```

Finally, the resetProduct mutation uses the spline method to remove the product at the desired index and adds a new product.

Account module

The account module is much simpler with a store holding just two booleans. These describe the logged in state and the show login form state. The actions and mutations are rather self explanatory so I'll leave it to you.

Testing the store

Without a complete frontend with the components to connect to the store yet, it is already possible to test the store functionality. We can use the developer tools for that! The store is exported in the components and Chrome/ff can select the <Root /> or <App /> component and refer to it by $root.

With the dispatch function available on the store we can test all actions. For example, adding a product to the store by executing:

```
$root.$store.dispatch('updateProduct')
```

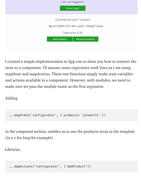
Note that we need to add the name of the module followed by a forward slash to let the store know it is the actions in module 'configurator'.

Mutations are also possible using commit:

```
$root.$store.commit('configurator/addProductAt')
```

However, a real dispatch is much more compared to the dev tools 😉

Connecting a component to the store



Sign up for free to join this conversation on GitHub. Already have an account? Sign in to comment

Sign up for free to join this conversation on GitHub. Already have an account? Sign in to comment

Now we need to add the name of the module followed by a forward slash to let the store know it is the actions in module 'configurator'.

Mutations are also possible using commit:

```
$root.$store.commit('configurator/addProductAt')
```

However, a real dispatch is much more compared to the dev tools 😉

Connecting a component to the store

Sign up for free to join this conversation on GitHub. Already have an account? Sign in to comment

Sign up for free to join this conversation on GitHub. Already have an account? Sign in to comment

I created a simple implementation in App.vue to show you how to connect the store to a component. I'll assume some experience with Vuex at I am using the mapGetters and mapActions helper functions to make code easy readable. Mutations are not used in this example. However, with modules, we need to make sure we pass the module name as the first argument.

Adding

```
...mapGetters('configurator', ['products', 'price'])
```

to the computed section, enables us to use the products array in the template (in a for loop for example):

Likewise,

```
...mapActions('configurator', ['addProduct'])
```

enables us to use 'addProduct' as a normal method and attach it to the add button.

Conclusion and next steps

In this post I have described the setup of the store including modules. The store functions could be tested with the dev tools of a browser, but a real component is needed to connect to the store.

In the upcoming post I'll start with the actual implementation of the postcard configurator. The mentioned parts in mainly here on erase the product component and build a top from simple components using v-model on custom components.

Stay tuned!

Originally published at https://www.blog.plint-sites.nl on May 15, 2019.

Vuejs · Vuex · Tutorial

10 claps

Written by
Pim Hoogheimstra · Follow
Lead by Scrum Lessons — Software Developer — Cloud Architect — Cloud Native — Mega Digital Alliance

Follow
PLint-sites
Dutch Tutorial Blog about Webdevelopment, Webapplications, Scrumcraft, Cloud Computing, Cloud Native, Mega Digital Alliance, Leadership and Other Interesting Stuff

Follow

More from this author

Vue.js 2 to 3 (Groot)ff PWA
Vuejs · Tutorial · 4min read

How to add push notifications to a progressive web app
Pim Hoogheimstra · Follow
Jun 2, 2019 · 4min read

Why I ditched Sublim
Pim Hoogheimstra · Follow
Jun 2, 2019 · 1min read

Responses

Write a response...

Conversation with Pim Hoogheimstra.

Jeremy Cox
May 31, 2019

Very article is a great read!

1 response

Pim Hoogheimstra
Jun 2, 2019

Thank you Anthony! Currently working on part #3 :-)