GeekBrains

Управление данными приложения с Vuex

Vue.js

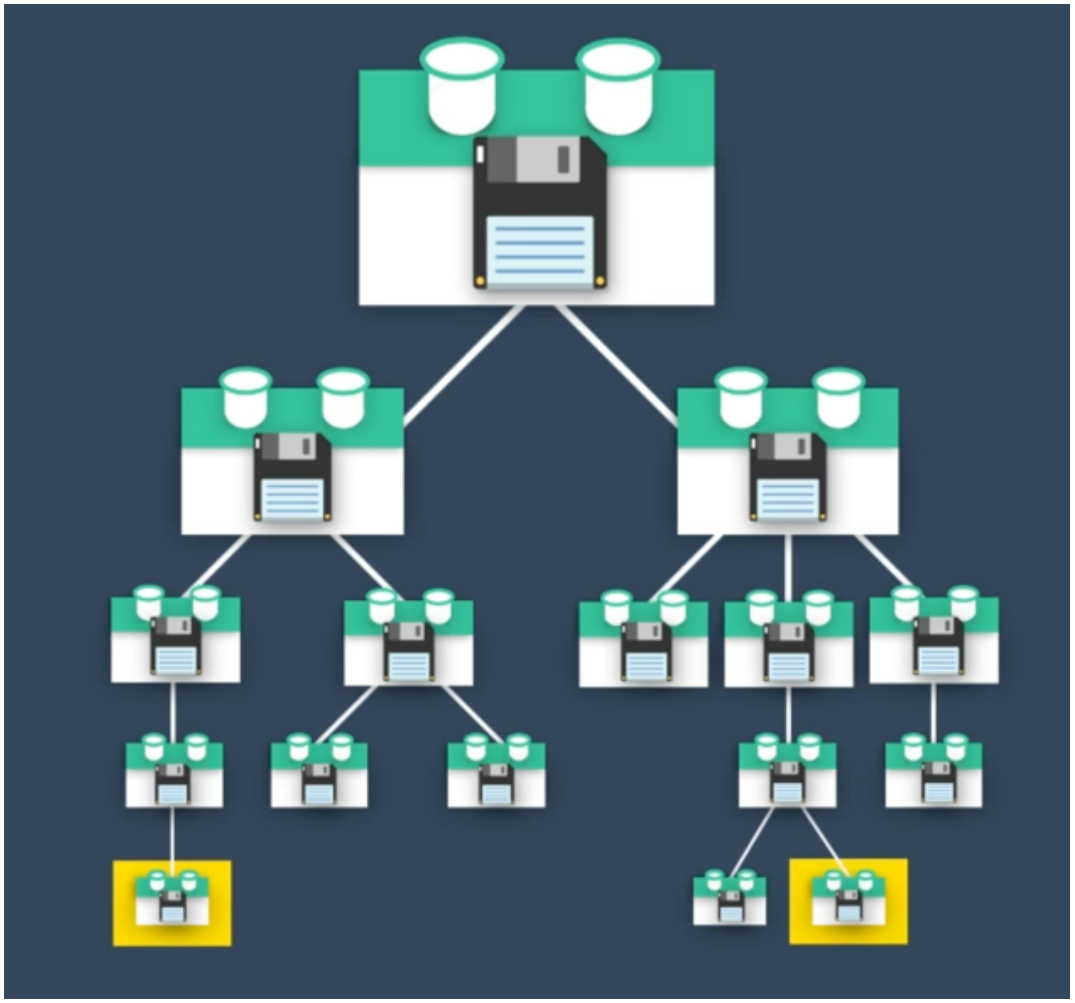
Оглавление

Данные в веб приложении – проблемы и решения.

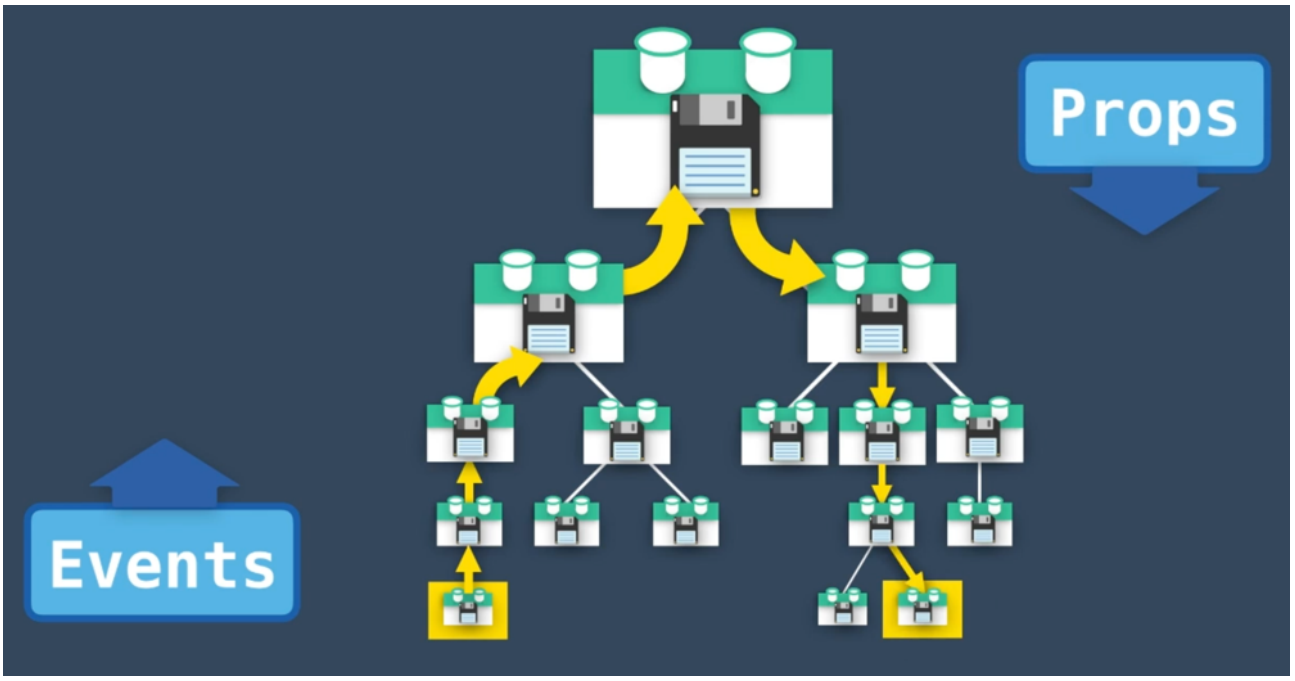
Давайте представим такую ситуацию. Мы проектируем какое-то веб-приложение. У нас есть несколько компонентов, которые умеют взаимодействовать друг с другом. Все работает как часы, код прост и понятен.

Приложение растет, развивается, нам необходимо расширить его функционал. Появляются новые модули, появляются новые источники данных для этих модулей. Наступает момент, когда вы обнаруживаете, что из простого и понятного кода образовалось нечто – куча связных компонентов, обмен данными происходит непонятно как, появилось большое количество props в родительских компонентах, которые этим компонентам и не нужны вовсе, а нужны каким-то далеким дочерним модулям.

Одной из основных проблем такой архитектуры является ситуация, когда 2 или более компонента/модуля используют одни и те же данные, и меняют их. Особенно сложно в случаях, когда эти компоненты находятся в совершенно не связанных друг с другом частях нашего приложения:



Если вспомнить наш прошлый урок, то мы знаем как во Vue можно передать данные от дочернего компонента родительскому и обратно: $emit и props. Если рассмотреть приведенную схему приложения, то, чтобы в нем передать информацию от крайнего левого компонента до крайнего правого, необходимо пройти весь путь через корень нашего приложения. Более того, сами данные должны храниться в этом корневом компоненте. Согласитесь – неудобно:



Такое приложение становится очень трудно поддерживать, все чаще появляются баги, увеличивается время для внесения чего-то нового. Мы задаемся логичным вопросом: как можно облегчить себе жизнь? Радует, что мы не первые, кто столкнулся с этой проблемой, и существуют подходы, которые помогут организовать нам работу с данными в большом приложении.

Прежде всего, почему бы нам не вынести данные, которые используются в различных частях нашего приложения, в какое-то отдельное хранилище? Доступ к этому хранилищу получат только те компоненты, которым действительно нужны данные. Данным действием мы сможем избавиться от ненужной цепочки посредников между двумя компонентами, которые выделены на схеме выше.

Давайте попробуем представить, что нам нужно получить от такого хранилища? Какими свойствами оно должно обладать и какие проблемы должно уметь решать?

1. Хранение данных. Первое и самое главное, что должно уметь делать хранилище – хранить данные и предоставлять к этим данным прямой доступ.
2. Изменение данных. Данные не обязаны быть константными значениями, а значит в процессе жизни нашего приложения они могут обновляться. Хранилище должно поддерживать изменение этих данных.
3. Актуальность состояния. Если данными из хранилища пользуются несколько компонентов, то при изменении этих данных, все причастные компоненты должны быть оповещены об этом. Должны предоставляться актуальные значения.

С тем, как должно выглядеть хранилище в нашем приложении, и какими свойствами оно должно обладать мы определились. В реализации же нам поможет библиотека Vuex.

Vuex. Основы.

Vuex – это официальная библиотека команды Vue для управления состоянием приложения. Под состоянием здесь понимается набор данных нашего приложения. То есть это тот «источник истины», на основании которого рисуется представление (компоненты) приложения. Библиотека реализует концепции Flux-архитектуры. Что это за концепции? Давайте рассмотрим их на примере. Чтобы это сделать, установим и подключим Vuex в наше приложение.

Давайте изменим наше приложение таким образом, чтобы его данные хранились не в корневом компоненте, а в хранилище Vuex.

Первым делом, установим саму библиотеку:

```
npm install vuex@3
```

Дальше необходимо непосредственно подключить ее к нам в приложение. Давайте создадим файл store/index.js с таким контентом:

store/index.js

```
import Vue from 'vue'
import Vuex from 'vuex'
Vue.use(Vuex)
export default new Vuex.Store({

})
```

Здесь мы сообщили Vue, что собираемся использовать Vuex в приложении, а также создали экземпляр глобального хранилища. Это созданное хранилище надо подключить в точке входа нашего приложения:

main.js

```
import Vue from 'vue'
import App from './App.vue'

Vue.config.productionTip = false

import store from './store'

```

```
new Vue({
 el: '#app',
 template: '<App />',
 components: {
   App,
 },
 store, // подключаем хранилище к нашему приложению
 render: h => h(App),
}).$mount('#app')
```

Итак, из чего у нас состоит хранилище Vuex? Давайте перечислим все директивы, из которых оно состоит.

1. В центре любого Vuex приложения находится специальный контейнер, в котором хранится состояние нашего приложения. Описывается этот контейнер в специальной директиве state.
2. Данные, которые хранятся в нашем хранилище нельзя изменять напрямую из любой точки нашего приложения. Это одна из концепций архитектуры, которая лежит в основе Vuex. Чтобы дать нам возможность управлять нашим состоянием, Vuex предоставляет специальные функции, которые это могут делать. Такие функции описываются в директиве mutations.
3. Помимо хранения и изменения данных, нам необходимо уметь получать наше состояние в любом месте (в любом компоненте) нашего приложения. Учитывая эту необходимость, Vuex предоставляет нам специальные функции (можно провести аналогию с computed свойствами компонентов), которые описываются в директиве getters.
4. Помимо перечисленных директив, существует еще одна группа функций, которая описывается в директиве actions. Функции, содержащиеся в данной директиве могут содержать бизнес-логику, относящуюся управлению хранилищем (например, получение данных).

index.js

```
import Vue from 'vue';
import Vuex from 'vuex';

Vue.use(Vuex);

export default new Vuex.Store({
 state: {
```

```
  // начальное состояние
 },
 mutations: {
  // методы для изменения состояния
 },
 actions: {
  // методы для асинхронных операций
 },
 getters: {
  // методы для чтения состояния
 },
 modules: {
  // модули Vuex для разделения хранилища на под-хранилища
 }
});
```

Давайте рассмотрим каждую из директив более подробно.

state

Состояние нашего хранилища. Можно сказать, что именно объект state и является тем самым хранилищем, о котором мы постоянно говорим – это тот объект, который содержит в себе данные, с которыми мы будем работать.

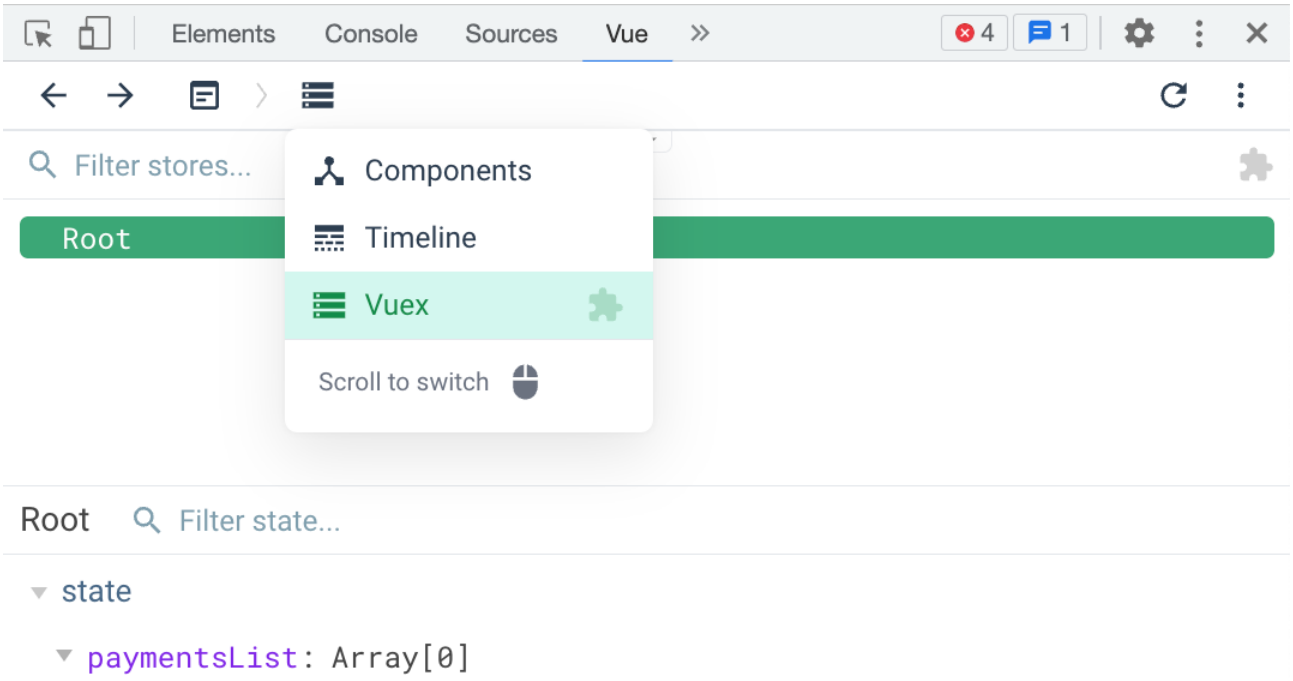
Если мы обратимся к корневому компоненту App.vue, то найдем там объявление свойства данных paymentsList. Сейчас в нем и хранятся все записи о наших расходах. Перенесем эти данные в state. Для тэста сразу заполним массив:

index.js

```
export default new Vuex.Store({
  state: {
      paymentsList: [],
  },
    . . .
})
```

Давайте посмотрим, к чему привели все наши наработки. Сохраним изменения и запустим наше приложение. Если мы откроем расширение Vue Dev Tools на вкладке

Vuex (справа от инспектора компонентов), то мы уже сможем найти наши объявленные данные:



На данной вкладке расширение Vue DevTools отображает все происходящие в приложении мутации, причем показывая в какое именно время была она сделана, и какие именно изменения произвела. Под списком прошедших мутаций мы можем увидеть текущее состояние нашего Vuex хранилища.

Отлично! Мы теперь умеем хранить данные в нашем Vuex хранилище. Пусть этими данными сейчас является пустой массив, но не все сразу. Как же можно заполнить данными хранилище? Здесь мы плавно переходим к основным концепциям библиотеки Vuex.

Vuex – наследник Flux-архитектуры.

Изначально Flux-архитектура – это подход для построения пользовательского интерфейса, основанный на однонаправленных потоках данных. Это значит, что накладываются ограничения на определенные операции с данными, в частности исключается возможность установки и обновления данных напрямую из компонентов. Для таких операций используется специальная точка входа внутри хранилища. Такой подход делает работу с глобальным состоянием более предсказуемым, так как позволяет проследить место и время изменения этого самого состояния.

Отсюда мы получаем первую концепцию: Хранилище – единственная сущность в приложении, которая знает, как изменить данные. Компонент может лишь запустить процесс изменения (добавление, обновление, удаление) данных, остальная работа остается за хранилищем.

Основываясь на выученной концепции, давайте создадим специальные функции в нашем хранилище. Во Vuex такие функции называются мутациями.

mutations

Мутации – единственный способ, который позволяет нам изменить состояние нашего хранилища. Это их основная и единственная задача. Давайте реализуем функционал, который будет помещать в paymentsList новый массив с объектами. Все мутации будем описывать в специальном объекте mutations:

index.js

```
mutations: {
   setPaymentsListData(state, payload) {
       state.paymentsList = payload
   },
}
```

Мутация является функцией, которая принимает в себя 2 аргумента:

- state - объект текущего состояния;
- payload – данные, передаваемые нашей мутации из компонента, совершающего мутацию. На основании этих данных хранилище будет менять свое состояние, переданное первым аргументом.

Давайте попробуем загрузить в наше хранилище начальные данные. Если мы обратимся к текущей реализации компонента App.vue, то увидим, что в хуке created у нас инициализируется свойство paymentsList из data этого компонента. Заменим данный код на вызов написанной мутации:

App.vue

```
 created() {
   this.$store.commit('setPaymentsListData',
this.fetchData())
 }
```

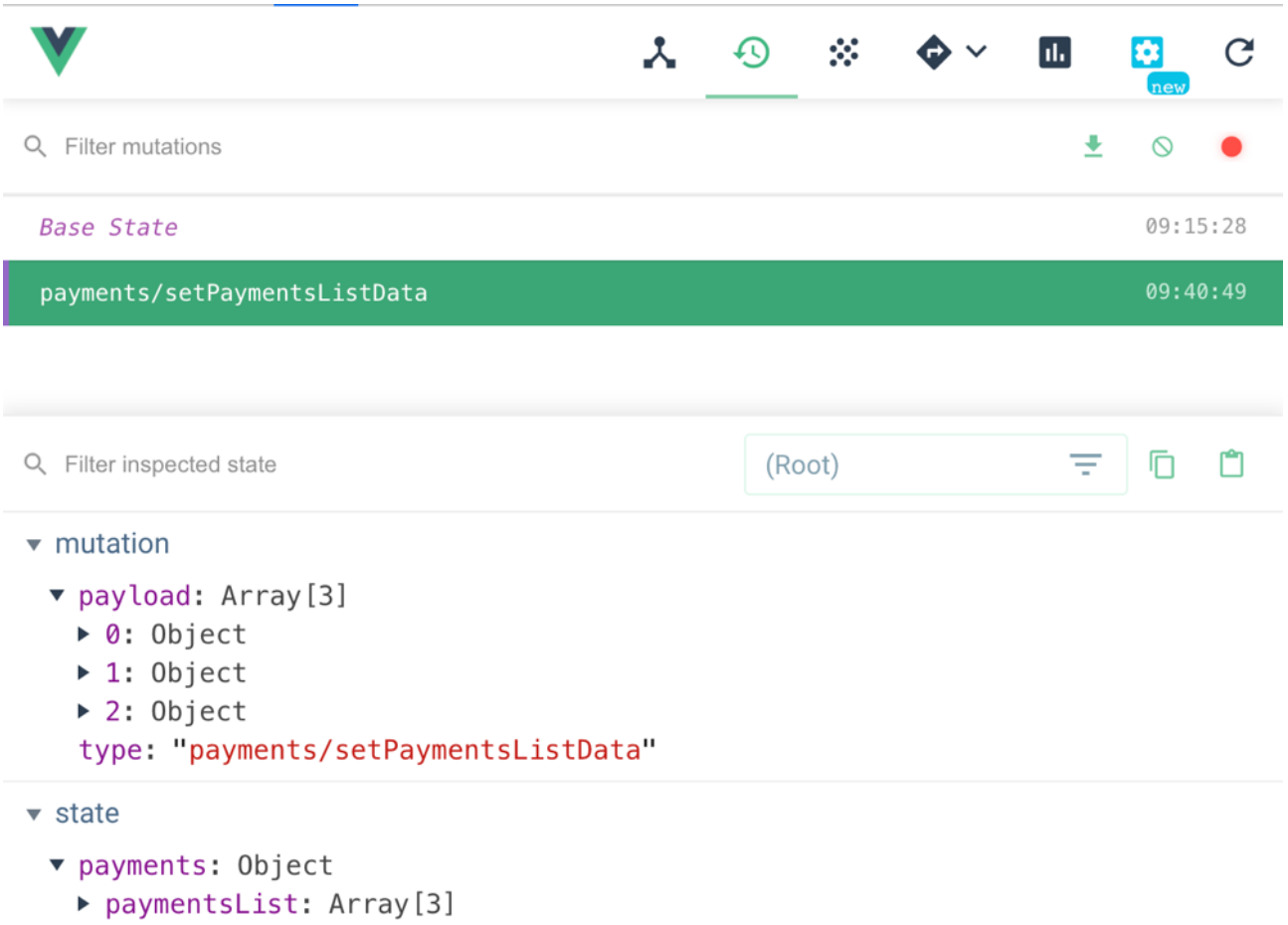
После того, как мы подключили библиотеку Vuex в приложение, а также подключили созданный экземпляр, мы получили возможность обратиться к глобальному объекту состояния через свойство $store.
Для запуска мутации мы не можем напрямую обратиться к функции-обработчику. Для этого нам необходимо воспользоваться специальным методом commit, который принимает в себя два параметра: имя мутации, и данные, которые будут использоваться внутри мутации для изменения состояния хранилища. В качестве данных у нас выступает результат работы функции fetchData().
Также, в нашем распоряжении имеется специальная функция mapMutations, входящая в состав библиотеки Vuex. С помощью этой функции можно проксировать мутации хранилища в методы компонента. Это значит, что мы можем обратиться к мутации setPaymentsListData напрямую, как к методу текущего компонента:

```
export default {
 …
 computed: {
   ...mapMutations([
     'setPaymentsListData',
   ])
 },
 created () {
   this.setPaymentsListData(this.fetchData())
 }
}
```

mapMutations принимает в себя перечисление названий тех мутаций, которые мы хотим использовать в нашем компоненте. Если есть необходимость использовать другое имя функции, нежели то, которое указано в хранилище, то мы можем воспользоваться объектным синтаксисом:

```
   ...mapMutations({
     updatePayments: 'setPaymentsListData'
   }),
```

Если мы сохраним текущую работу и запустим наше приложение, то во Vue Dev Tools сможем увидеть следующую картину:



Мы видим, что у нас создалась запись об изменении состояния нашего хранилища: была вызвана мутация типа setPaymentsListData, в которую была передана «полезная нагрузка» в виде payload. Ниже мы можем заметить, что текущее состояние больше не является пустым массивом, а в нем находятся 3 элемента.

Отлично! Мы научились создавать хранилище и записывать в него данные. Но, как же мы можем воспользоваться этими данными внутри компонентов? Раз мы реализовали так называемые «сеттеры»-мутации, значит есть и «геттеры»? Совершенно верно!

Vuex – получение данных в компоненте.

getters

Vuex может предоставлять внешнему приложению доступ к своему состоянию (объекту state) напрямую. Чтобы обратиться к состоянию мы можем воспользоваться уже знакомым нам глобальным свойством $store, в котором имеются необходимые данные:

```
this.$store.$state
```

Однако, обращаться к состоянию напрямую - плохая практика. Если у нас в приложении много обращений, и в какой-то момент возникла необходимость изменить структуру нашего объекта состояния, то придется менять обращения везде в коде, где они встречались. Было бы намного удобнее, если мы могли получать данные через какой-нибудь прокси. Такой прослойкой между компонентом и хранилищем выступают getters.

getters, или как мы будем их называть «геттеры» – это по своей сути computed свойства нашего модуля vuex. Они позволяют нам получить данные, основанные на текущем состоянии (state).

Мы можем реализовать в геттерах те свойства, которые нам впоследствии могут понадобиться в компонентах нашего приложения. Если к этим свойствам будут обращаться несколько компонентов, то таким образом мы можем избавимся от дублирования кода.
Давайте реализуем 2 геттера, которые будут отдавать:

1. Массив с нашими платежами таким, какой он есть.
2. Сумму всех платежей в массиве. Сразу оговорим, как будем это делать.

Массив состоит из объектов вида:

{ date: ’28.03.2020’, category: ‘Food’, value: 169 }

В данном объекте свойство value определяет стоимость платежа. Чтобы получить полную стоимость достаточно пройтись по массиву и сложить все эти значения.

store/index.js

```
getters: {
   // получаем список paymentsList
   getPaymentsList: state => state.paymentsList,

   // получаем суммарную стоимость всех платежей
   getFullPaymentValue: state => {
       return state.paymentsList
           .reduce((res, cur) => res + cur.value, 0)
```

```
    },
}
```

Можете обратить внимание, что геттер представляет из себя функцию, которая принимает один параметр state – текущее состояние. С помощью этого параметра мы обращаемся к нашему хранилищу, а именно к его данным. Как и computed свойства, геттер обязан возвращать результат через оператор return.

Давайте обратимся к компоненту PaymentsDisplay.vue и получим в нем данные из нашего хранилища. Аналогично тому, как мы обращались к мутациям, мы можем обратиться к геттерам через глобальное свойство $store. В этом случае с геттерами мы можем обратиться к ним напрямую, не используя никакую дополнительный метод,
Воспользоваться следующим кодом:

PaymentsDisplay.vue

```
<template>
  <div>
    {{ getFPV }}
  </div>
</template>

<script>
…
export default {
  …
  computed: {
    getFPV () {
      return this.$store.getters.getFullPaymentValue
    }
  },
}
</script>
```

Также, аналогично с мутациями, в нашем распоряжении имеется специальная функция mapGetters. С помощью этой функции можно проксировать геттеры хранилища в вычисляемые свойства компонента. Это значит, что мы можем обратиться к свойству getFullPaymentValue напрямую, как к computed свойству текущего компонента:

App.vue

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

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

Давайте доработаем наше приложение таким образом, чтобы форма добавления новых платежей также работала с Vuex. Обратимся к компоненту AddPaymentForm.vue.

Сейчас при отправке данных формы (при клике на кнопку Save!) у нас запускается метод onSaveClick. В этом методе генерируется событие addNewPayment, которое сообщает родительскому компоненту, что мы добавили какие-то данные, а также передаем эти самые данные родительскому компоненту. Сейчас в App.vue мы больше не храним текущее состояние, поэтому надобности в отправке событий у нас больше нет. Реализуем соответствующую мутацию в хранилище:

store/index.js

```
const mutations = {
    ...
    addDataToPaymentsList(state, payload) {
        state.paymentsList.push(payload)
    }
 }
```

Теперь можем воспользоваться ей в компоненте формы:

AddPaymentForm.vue

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

export default {
  data () {…},
  computed: {
    ...mapMutations([
      'setPaymentsListData',
    ]),
```

```
    getCurrentDate () {…}
  },
  methods: {
    onSaveClick () {
      const data = {
        amount: +this.amount,
        type: this.type,
        date: this.date || this.getCurrentDate,
      }
      this.$emit('addNewPayment', data)
    }
  }
}
</script>
```

Отлично! Мы добавили возможность обновлять наше хранилище Vuex из компонента формы. Как же теперь нам сообщить компоненту PaymentsDisplay.vue, что хранилище обновлено и надо получить новые данные для перерендера?

Если мы запустим наше приложение и попробуем добавить платеж через форму, то мы увидим, что список платежей сразу изменится, с поступлением новых данных. Это значит, что больше никаких операций с нашей стороны реализовывать не требуется. Данные, которые мы получаем из хранилища с помощью геттеров и устанавливаем с помощью мутация являются реактивными. Это еще одна ключевая концепция, которая наследуется от Flux-архитектуры, лежащей в основе Vuex: хранилище сообщает представлению (компонентам) об изменении своего состояния, не передавая при этом данные напрямую (не вызывая никаких коллбэков).

Мы научились хранить, изменять и получать данные из нашего хранилища. Полученных знаний, казалось бы, вполне достаточно для полноценной работы с Vuex Store. Однако, есть еще один аспект, который мы не рассмотрели. Перед обновлением данных в хранилище часто возникает потребность получить эти данные. Например, обращаясь за ними к бэкэнд серверу. Получать данные хочется в рамках хранилища, так как эта логика полностью укладывается в работу с данными. Где можно произвести такой запрос? Непосредственно в мутации этого сделать нельзя, так как мутация – синхронная операция и выполнение в ней любых асинхронных запросов запрещено.

На помощь нам придет еще одна директива, которую мы определяем в хранилище – actions, действия.

Vuex – работа с асинхронными запросами.

actions

Actions – действия. Это специальные функции, которые содержат бизнес-логику приложения в рамках хранилища. Эти функции НЕ могут изменять состояние. То есть напрямую на данные они не влияют. Вызов действия является асинхронной операцией (то есть при вызове actions исполнение нашего кода не прекратится). Это удобно и зачастую именно в действиях выполняют запросы к серверу, на основании которых могут меняться данные нашего приложения. Не стоит выполнять такие сетевые вопросы напрямую из компонента.

Давайте перенесем функцию fetchData из компонента App.vue в наше хранилище. Также, давайте сымитируем асинхронность данной функции, как будто она делает запрос на сервер во время своего выполнения. Результатом работы функции будет объект Promise, который будет указывать на завершенность работы. «Получив» данные, функция должна положить их в хранилище.

index.js

```
    actions: {
        fetchData({ commit }) {
            setTimeout(() => {
                const paymentsList = [
                    {
                        date: '28.03.2020',
                        category: 'Food',
                        value: 169,
                    },
                    {
                        date: '24.03.2020',
                        category: 'Transport',
                        value: 360,
                    },
                    {
```

```
                    date: '24.03.2020',
                    category: 'Food',
                    value: 532,
                },];
            commit('SET_PAYMENT', paymentsList);
        }, 1000);
    }
}
```

Как мы видим, функция-action в качестве первого параметра получает объект контекста хранилища. Для модулей, зарегистрированных в рамках своего собственного пространства имен (namespaced модули), этот контекст ограничивается текущим модулем.
Контекст содержит в себе те же методы и свойства, что и сам экземпляр хранилища, поэтому мы можем вызвать context.commit для инициирования мутации, можем обратиться к геттерам через context.getters, а также обратиться напрямую к состоянию через context.state.

Вторым аргументом actions, по аналогии с мутациями получают «полезную нагрузку» - payload. Данные, полученные в payload, могут использоваться в качестве параметров к запросу на сервер, либо эти данные могут участвовать непосредственно в последующей мутации.

Используемая литература

1. Официальный сайт Vue.js - ссылка
2. Документация Vue CLI - ссылка