GeekBrains

Взаимодействие между компонентами

Vue.js

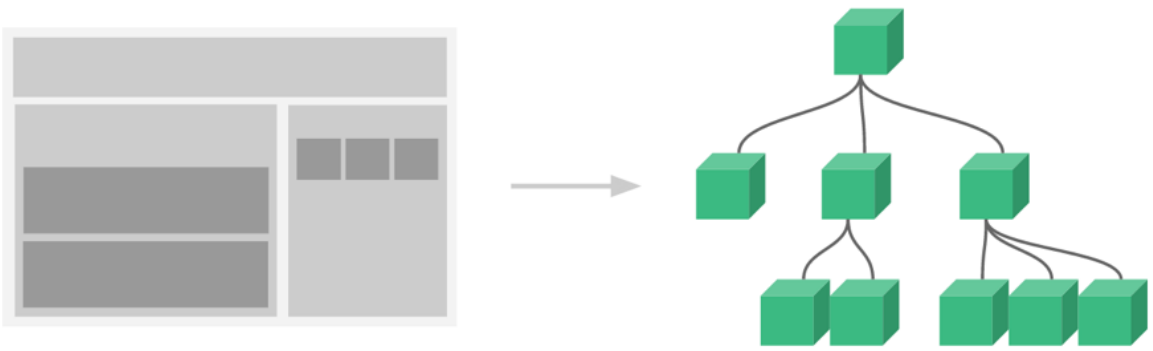
Оглавление

Введение

Первым делом давайте освежим в памяти что такое компонент. Компонент – логически независимая часть приложения, которая отвечает за обработку и визуализацию конкретной задачи. Композиция таких независимых частей должна представлять из себя рабочий функционал.

Можем обратиться за примером из реальной жизни. Представим, что у нас есть проект – построить дом. На какие логические составляющие можно разбить наш конечный результат? Стена, крыша, окно, дверь – это отдельные, несвязанные между собой, но логически завершенные кусочки целого проекта. Они имеют свой облик и предоставляют нам специфичный для себя набор функционала. Собрав эти части и связав между собой, мы получим то, к чему стремились – проект готового дома.

Вернемся к ближе к нашей теме. По аналогии с предыдущим примером, основным строительным блоком наших приложений является компонент. Мы можем визуально разбить нашу страницу на множество частей: шапка, блок с информацией, меню, кнопки. Все эти части могут быть выделены в отдельные компоненты. Особенно это важно для тех частей, которые повторяются множество раз в нашем приложении. Выделив их в компоненты, мы сможем избежать дублирования кода.
Давайте рассмотрим пример.



На данной схеме мы видим типичный интерфейс информационного сайта. У нас есть:

· заголовок
· центральный блок, в котором списком идут отдельные блоки новостей
· колонка справа, в которой перечислены квадратиками пользователи, товары или какие-то другие элементы

Все это можно и нужно выделить в отдельные компоненты.

Что такое компонент и для чего он нужен мы разобрались. Осталось понять, что этот компонент представляет из себя внутри. Как мы уже видели на прошлых занятиях, компонент – это блок кода, который может включать в себя описание видимой

части – html верстку, а также логику на JavaScript, которая управляет взаимодействием пользователя с данной версткой.
Одним из удобств компонентного подхода является способ подключения и использования компонентов в нашем коде. Современные фреймворки, включая изучаемый нами Vue, позволяют регистрировать свои html-тэги, которые и будут отображать созданные нами компоненты. Комбинируя между собой простые компоненты, мы можем легко строить самые сложные интерфейсы.

Благодаря этому, при наличии необходимых компонентов, основу сайта из примера выше можно было бы описать всего в нескольких строках кода:

```
<div>
 <Header />
 <NewsBlock />
 <RightColumn />
</div>
```

Выглядит просто. Здесь важно будет упомянуть, что компоненты – это переиспользуемые экземпляры Vue со своим именем. Это значит, что в каждом компоненте у нас доступны те же свойства, что доступны при создании нового экземпляра Vue: data, methods, computed и все остальные. У каждого компонента свой собственный набор данных свойств. То есть, например, из одного компонента нельзя обратиться к методам другого компонента.

Теперь рассмотрим следующий пример. У нас есть простой компонент кнопки со счетчиком:

```
<template>
 <button @click="onClick">Clicked {{ counter }} times!</button>
</template>
<script>
export default {
 data() {
   return {
     counter: 0,
   }
 },
 methods: {
   onClick() {
```

```
      this.counter++
    }
  }
}
</script>
```

Давайте добавим несколько таких кнопок себе на страницу.

```
<template>
 <div id="app">
   <CounterButton />
   <CounterButton />
   <CounterButton />
 </div>
</template>
```

Если запустить данный пример и нажать на одну из кнопок, то мы увидим, что поменяется состояние только той кнопки, на которой был произведен клик. Отсюда делаем явный вывод: при каждом новом использовании компонента создается новый экземпляр этого компонента со своим собственным состоянием и со своим жизненным циклом. Давайте подробнее рассмотрим что это такое.

Жизненный цикл.

В тот момент, когда мы решили использовать написанный нами компонент, он начинает свой жизненный цикл. Его можно описать тремя словами: создание, изменение, удаление. Каждый этап жизненного цикла можно отследить, и при наступлении нового этапа можно выполнять необходимые нам операции. Делается это с помощью специальных функций, которые называются хуками жизненного цикла. Давайте рассмотрим этот процесс более подробно.

Представим, что рендер Vue дошел до объявления нашего компонента в шаблоне:

```
<template>
 ...
 <CounterButton />
 ...
</template>
```

Начинается инициализация компонента CounterButton – «эпоха» создания. В рамках данного промежутка времени компонент будет создан и встроен в DOM. К этапу создания компонента можно отнести 4 хука жизненного цикла: beforeCreate, created, beforeMount, mounted. Можно использовать их, чтобы настроить компонент (проинициализировать начальные значения в data, начать выполнять запрос на сервер, для получения необходимых данных и так далее).

beforeCreate

Этап создания начинается с хука beforeCreate. На данном этапе данные еще не стали реактивными, события не настроены. Например, если в рамках хука beforeCreate попытаться изменить свойство в counter, то после отображения компонента мы увидим, что нас постигла неудача:

```
<template>
 <button @click="onClick">Clicked {{ counter }} times!</button>
</template>



<script>
export default {
 data() {
   return {
     counter: 0,
   }
 },
 beforeCreate() {
   this.counter = 1
 },
}
</script>
```

Выведет Clicked 0 times!

created

На данном этапе уже можно получить доступ к реактивным данным. Если в примере выше заменить хук beforeCreated на created, то после отображения мы увидим ожидаемую единицу. Верстка и виртуальный DOM все еще не готовы.

beforeMount

Начало этапа отображения нашей верстки. Начинается этот этап с хука beforeMount. Вызывается он непосредственно перед первой отрисовкой. Шаблон уже скомпилирован, но еще не встроен в DOM. В рамках этого хука уже можно получить доступ к корневому элементу нашего компонента:

```
beforeMount () {
  console.log('Component root element: ', this.$el)
},
```

mounted

В хуке mounted мы получаем полный доступ к компоненту. Все проинициализировано, все отображено, все работает.
В рамках данного хука мы можем повесить свои события на элементы. Ради примера давайте повесим событие mouseover на нашу кнопку:

```
<template>
 <button>Hover!</button>
</template>
<script>
export default {
 methods: {
   onMouseOver() {
     console.log('Hovered on button!')
   }
 },
 mounted() {
   const btn = this.$el.querySelector('button')
   if (btn) {
     btn.addEventListener('mouseover', this.onMouseOver)
   }
 }
}
</script>
```

Теперь при наведении у нас в консоли будет появляться соответствующее сообщение.

В таком состоянии компонент может просуществовать все оставшееся время, если мы сами не будем изменять его свойства. Если все же будем, то мы сможем воспользоваться еще двумя хуками. Наступает «эпоха» изменений!

beforeUpdate

Хук beforeUpdate выполняется после изменений данных в компоненте, непосредственно перед перерисовкой DOM. В нем можно получить новое состояние всех данных, перед их отображением.

updated

Хук updated выполняется после перерисовки DOM. Если есть необходимость обратиться к DOM после актуализирования данных, то лучше всего делать это в рамках данного хука.

В нашем мире ничего не вечно, ровно как и срок жизни компонента. Когда пользователь переходит на новую страницу, где нашего компонента больше не будет, или компонент скрывают под директивой v-if, начинается процесс уничтожения этого компонента. Как вы могли заметить, все хуки делятся на 2 этапа before и after. Процесс уничтожения компонента не исключение, и vue нам также предлагает 2 хука: beforeDestroy и destroyed.

beforeDestroy

В рамках «эпохи уничтожения» следует «нейтрализовать продукты жизнедеятельности» компонента. В одном из примеров выше мы явно навесили обработчик события наведения курсора мыши на кнопку в хуке mounted. При уничтожении компонента такие обработчики надо удалять.

```
beforeDestroy() {
  const btn = this.$el.querySelector('button')
  if (btn) {
    btn.removeEventListener('onmouseover', this.onHover)
  }
```

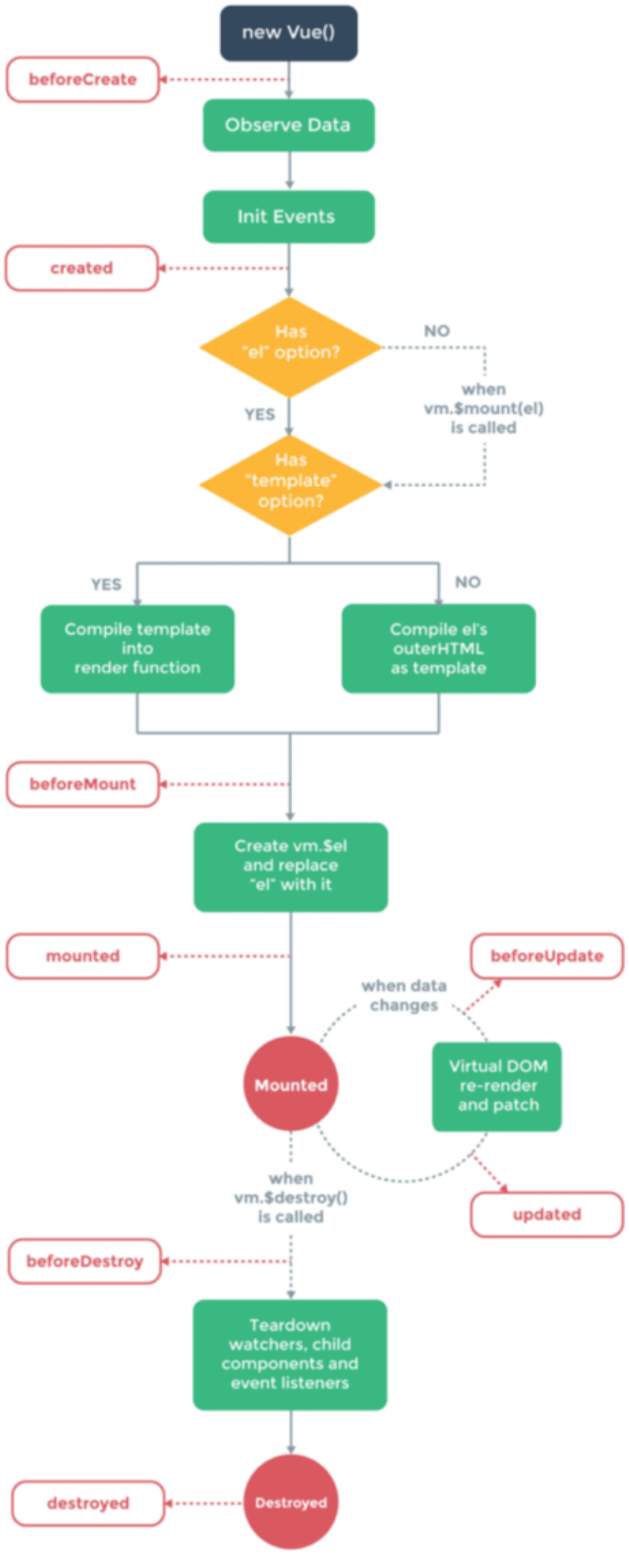
```
},
```

beforeDestroy выполняется непосредственно перед удалением компонента из DOM.

destroyed

К тому моменту, как мы добрались до хука destroyed, от нашего компонента мало что осталось. Всё, что было к нему прикреплено, уже уничтожено, сам компонент удален из DOM. Можно использовать данный хук, чтобы, например, проинформировать сервер об удалении компонента, отправив соответствующую аналитику.

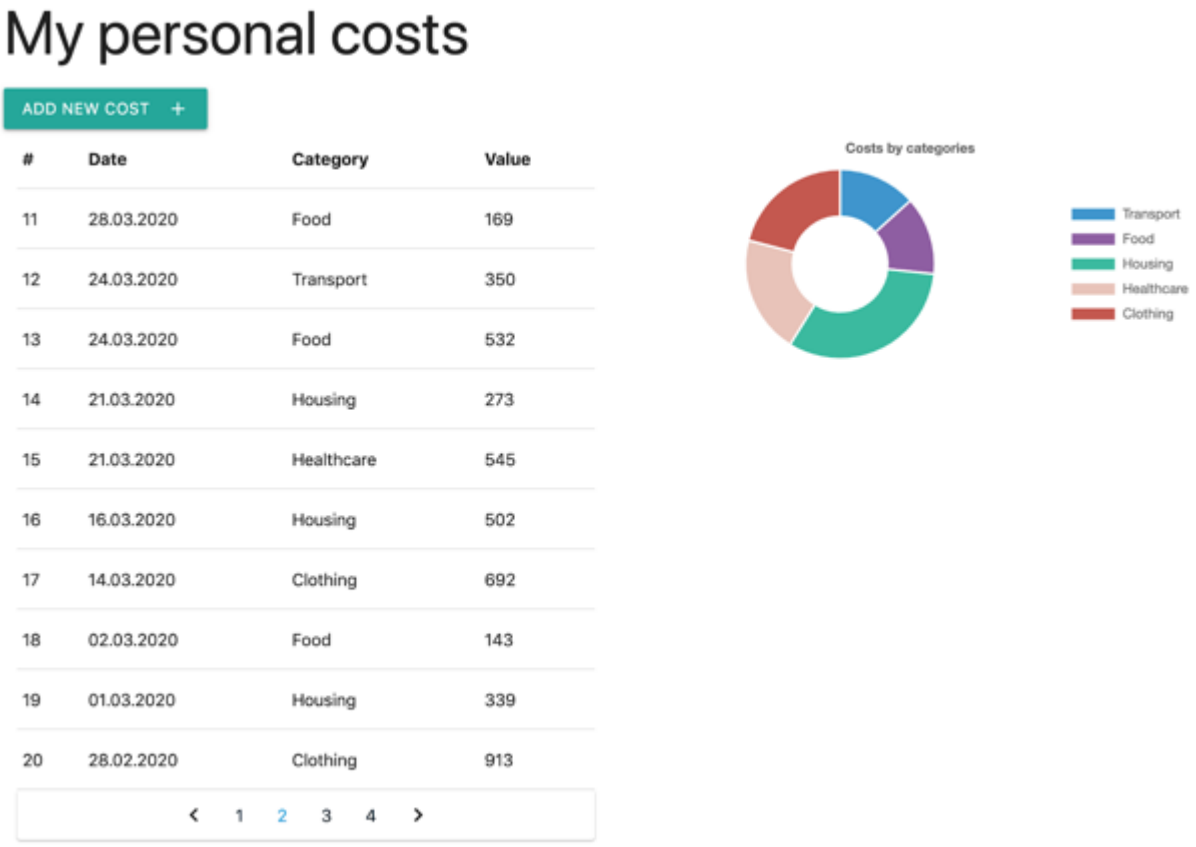
На схеме ниже мы можем увидеть все рассмотренные состояния нашего компонента:


new Vue()
beforeCreate
Observe Data
Init Events
created
Has "el" option?
NO
when vm.$mount(el) is called
YES
Has "template" option?
YES
NO
Compile template into render function
Compile el's outerHTML as template
beforeMount
Create vm.$el and replace "el" with it
mounted
beforeUpdate
when data changes
Mounted
Virtual DOM re-render and patch
when vm.$destroy() is called
updated
beforeDestroy
Teardown watchers, child components and event listeners
destroyed
Destroyed

Практическая часть

Приложение для ведения личных финансов.

Мы узнали, что такое компонент и рассмотрели его базовую работу. Давайте применим полученные знания на практике.



С помощью нашего приложения мы сможем:

· добавлять свои расходы с помощью специальной формы;

· просматривать добавленные расходы с помощью таблицы с пагинацией;

· отслеживать процентное соотношение категорий наших расходов с помощью удобного графика

Так же, как мы делали раньше, давайте визуально разобьем наше будущее приложение на отдельные части. Смотря на предложенный вариант работы выше, что мы можем выделить? Начнем разбивать наше приложение поэтапно, то есть просматривая макет сверху вниз, слева направо.

Первое, что нам встречается (если не считать заголовок нашего приложения) – это кнопка добавления новой статьи расходов. Кнопка – это такой элемент на странице, который точно может быть переиспользован в будущем, так как кнопок в приложении может быть большое количество. Явный кандидат на отдельный компонент.

По нажатию на кнопку будет появляться сама форма добавления новых расходов. Это самостоятельная логическая единица нашего проекта. Будет целесообразно выделить логику формы в отдельный компонент, что мы и сделаем.

Последнее что осталось – отображение ранее добавленной информации. Таблица с записями и график. Тоже выделим их в отдельные компоненты.

Давайте подведем итог, какие компоненты нам придется с вами реализовать:
1) Кнопка (Button)
2) Форма добавления расходов (AddPaymentForm)
3) Отображение расходов (PaymentsDisplay)

В будущем, в ходе разработки мы можем разбить эти компоненты на еще более мелкие, для удобства.

Создаем приложение.

Определившись с планами, можем начинать писать наше приложение. Предположим, что у нас в распоряжении находится настроенная среда webpack, и у нас есть возможность работать с однофайловыми компонентами. Тогда, первым действием, создадим точку входа для нашего приложения, в которой инициализируем Vue экземпляр – так же, как делали это на первом занятии:

main.js

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

Vue.config.productionTip = false

new Vue({
 render: h => h(App),
}).$mount('#app')
```

В качестве элемента, где мы будем создавать наше Vue приложение, мы указали блок с id=”app”. Поэтому в index.html, куда будем подключать наш скрипт, надо не забыть этот элемент создать.

Шаблоном нашего приложения будет выступать компонент App. Данный компонент будет являться агрегатором нашего приложения, то есть именно он будет в себя включать те компоненты, на которые мы предварительно разбили наше приложение. Давайте создадим его:

App.vue

```
<template>
 <div id="app">
   <div>
     <header>
       <div>My personal costs</div>
     </header>
     <main>
       <PaymentsDisplay />
     </main>
   </div>
 </div>
</template>
<script>
import PaymentsDisplay from './components/PaymentsDisplay'
export default {
 name: 'App',
 components: {
   PaymentsDisplay,
 },
}
</script>
<style>
</style>
```

Пока обратим свое пристальное внимание на реализацию основной идеи нашего приложения: отображение расходов. За это будет отвечать компонент

PaymentsDisplay. Кнопку, которая будет скрывать и раскрывать форму, а также саму форму реализуем чуть позднее.

Начинаем работу над компонентом PaymentsDisplay:

```
<template>
   <div>
       <!-- Хотим тут отобразить данные -->
   </div>
</template>
```

Учитывая функционал нашего компонента – отображать данные, а не хранить их, мы сталкиваемся с проблемой. Откуда нам брать эти данные? Мы их должны получить откуда-то извне. В нашем случае этим «вне» будет являться родительский компонент App.vue.
С источником данных определились, осталось наладить связь между компонентом App.vue и компонентом PaymentsDisplay. С такой задачей нам помогут справиться props.

Создание Props для компонентов

При создании компонента App.vue мы будем запрашивать данные о ранее зафиксированных расходах. Запрошенные данные поместим в свойство paymentsList в блоке data. Сделать это можно, использовав ранее изученный хук created.

Функцию fetchData пока реализуем таким образом, чтобы она отдавала какой-то определенный набор тестовых данных.

App.vue:

```
import PaymentsDisplay from './components/PaymentsDisplay'

export default {
 name: 'App',
 components: {
   PaymentsDisplay,
 },
 data() {
```

```
    return {
      paymentsList: [],
    }
  },
  methods: {
    fetchData() {
      return [
        {
          date: '28.03.2020',
          category: 'Food',
          value: 169,
        },
        {
          date: '24.03.2020',
          category: 'Transport',
          value: 360,
        },
        {
          date: '24.03.2020',
          category: 'Food',
          value: 532,
        },
      ]
    },
  },
  created() {
    this.paymentsList = this.fetchData()
  },
}
```

Мы хотим отобразить полученный список. Для этого, свойство paymentsList необходимо передать в компонент PaymentsDisplay. Как ранее уже было сказано, сделать это можно с помощью props.

Props – это список входных параметров, по которым разрешено получение данных из родительского компонента. Можно использовать простой синтаксис в виде массива названий параметров, или объект, который предоставляет дополнительные возможности: проверку типов, валидацию данных и значения по умолчанию.

Отсюда следует, что в компоненте PaymentsDisplay нам необходимо указать свойство, по которому можно получить наши данные. Назовем это свойство items:

PaymentsDisplay.vue:

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

<script>
export default {
   name: 'PaymentsDisplay',
   props: ['items'],
}
</script>
```

Такая запись говорит нам, что свойство items является «аргументом» нашего компонента. Если передать в это свойство данные, то мы сразу можем их использовать в шаблоне. Осталось разобраться, как же именно можно передать данные из родительского компонента в свойство items.

Для передачи используется следующий синтаксис.

app.vue

```
<template>
 ...
  <PaymentsDisplay :items="paymentsList" />
 ...
</template>
```

Валидация props

Чуть ранее, в определении значения props, мы упомянули, что помимо простого синтаксиса массива, в описании props можно использовать расширенный синтаксис

в виде объекта. Данный синтаксис позволяет нам валидировать значения, приходящие извне.

Зачем это нужно? Если ответить просто – для упрощения жизни программисту. Может быть ситуация, когда по ошибке в значение props передается свойство, не предназначенное этому props. В таком случае может возникнуть неопределенное поведение в программе. Скорее всего, просто все перестанет работать, и не будет понятно, почему. Если мы сделаем проверку принимаемых параметров, и придут не те данные, которые мы ожидаем, то Vue покажет соответствующее предупреждение в консоли, что значительно повысит скорость нахождения ошибки.

Давайте перепишем описание свойства со списком элементов в более строгом виде с указанием начального значения:

PaymentsDisplay.vue

```
  props: {
      items: {
          type: Array,
          default: () => {
              return []
          }
      }
  },
```

Теперь, если мы вызовем наш компонент, например, таким образом:

```
    <PaymentsDisplay :items="'any list data'" />
```

У нас возникнет ошибка, что ожидался массив, а пришла строка.
Мы смогли вывести данные в компоненте PaymentsDisplay, поэтому пришло время вспомнить о компоненте AddPaymentForm. Напомним, что этот компонент будет выводить форму для добавления новой записи наших расходов.

Для начала давайте создадим верстку формы и научим ее сохранять введенные значения в блок данных компонента:

AddPaymentForm.vue:

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

```
        <input placeholder="Amount" v-model="amount" />
        <input placeholder="Type" v-model="type" />
        <input placeholder="Date" v-model="date" />
        <button>Save!</button>
    </div>
</template>

<script>
export default {
    data() {
        return {
            amount: '',
            type: '',
            date: '',
        }
    }
}
</script>
```

Отлично! Осталось только передать сохраненные в блоке data данные в родительский компонент App.vue. И снова перед нами встает проблема – как это можно сделать? Воспользоваться props в этом случае не получится, так как с их помощью можно передать значение от родительского компонента дочернему, и никак не наоборот. При попытке изменить значение props в дочернем компоненте у нас возникнет ошибка - Avoid mutating a prop directly since the value will be overwritten whenever the parent component re-renders. Так делать нельзя!

На помощь нам приходят события!

$emit – генерация событий и передача данных

Любой компонент может в себе генерировать событие, на которое родительский компонент может подписаться. С помощью события дочерний компонент сообщает родителю, что произошло какое-то событие, и вместе с этим сообщением можно передать любые данные. Давайте попробуем сгенерировать событие добавления новой записи в компоненте нашей формы. Генерировать событие будем по нажатию на кнопку Save!

AddPaymentForm.vue:

```vue
<template>
    <div>
        <input placeholder="Amount" v-model="amount" />
        <input placeholder="Type" v-model="type" />
        <input placeholder="Date" v-model="date" />
        <button @click="onSaveClick">Save!</button>
    </div>
</template>

<script>
export default {
    data() {
        return {
            amount: '',
            type: '',
            date: '',
        }
    },
    computed: {
        getCurrentDate() {
            const today = new Date();
            const d = today.getDate()
            const m = today.getMonth() + 1
            const y = today.getFullYear()
            return `${d}.${m}.${y}`
        }
    },
    methods: {
        onSaveClick() {
            const data = {
                amount: +this.amount,
                type: this.type,
                date: this.date || this.getCurrentDate,
            }
            this.$emit('addNewPayment', data)
```

```
        }
    }
}
</script>
```

Как это работает?

1. Мы повесили функцию onSaveClick на нативное событие @click.
2. При клике, внутри этого метода мы формируем объект новой статьи расходов на основании данных, введенных в форму.
3. Генерируем событие с помощью вызова $emit. В параметры вызова $emit передаются название события, которое мы хотим сгенерировать, а также данные, которые будут доступны в обработчике этого события. Давайте в этом убедимся.

На стороне App.vue необходимо обработать событие, сгенерированное дочерним компонентом AddPaymentForm. Как можно повесить обработчик такого кастомного события? Точно так же, как и повесить обработчик на нативное событие, что мы уже умеем делать:

App.vue

```
<template>
 ...
  <AddPaymentForm @addNewPayment="addNewPayment" />
 ...
</template>

<script>
import AddPaymentForm from './components/AddPaymentForm';

export default {
 ...
 methods: {
  ...
  addNewPayment (data) {
   this.paymentsList = [...this.paymentsList, data]
  }
 },
 ...
}
</script>
```

Когда мы используем функцию-обработчик на кастомном событии, в аргументах этой функции мы можем обратиться к данным, которые были переданы вместе с событием. В нашем случае, в аргументе data мы будем иметь доступ к объекту, созданном на основании данных формы в компоненте AddPaymentForm. Таким образом мы смогли передать данные из AddPaymentForm в компонент App, а из App в компонент PaymentsDisplay.

Итоги урока

Конечно давайте собирать всё воедино

PaymentDisplay.vue

```
<template>
 <div>
   <div v-for="item in items" :key="item.id">
     <div class="content">
       <div class="content__item">{{ item.date }}</div>
       <div class="content__item">{{ item.category }}</div>
       <div class="content__item">{{ item.value }}</div>
     </div>
   </div>
 </div>
</template>
```

Стили можно задать по желанию

```
<style scoped>
.content {
 display: grid;
 grid-template-columns: repeat(3, 1fr);
}
```

```
.content__item {
 padding: 8px 16px;
 border: 1px solid #ccc;
}
</style>
```

Используемая литература

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