GeekBrains

Встроенные директивы и работа с данными в компоненте

Vue.js

Оглавление

Введение

Мы поговорим о том, как устроена реактивность во Vue, чтобы лучше понимать, как работать с данными в шаблонах и выстраивать иерархию компонентов.
Посмотрим на OptionsAPI в целом.
Рассмотрим самые важные директивы v-bind и v-model подробнее. И, наконец, научимся слушать регистрировать события на странице и реагировать на них.

На этом занятии мы продолжим доработку калькулятора который начали на первом занятии. Мы добавим к нему следующий функционал:

- возможность проверки данных
- сохранение истории операций
- и оптимизируем код с использованием новых концепций.

Практическая часть

Данный урок мы начнём с калькулятора, который удалось реализовать на предыдущем уроке, он отлично подходит для реализации простого и быстрого подхода, но наверняка заметили что там используется метод eval() который не является отличным решением, для чего-то более сложного. Как итог, нам необходимо написать калькулятор, который может выполнять не только базовые функции, но еще и другие математические задачи

Первым делом вам потребуется убедиться что у вас установлен node.js для этого вам необходимо открыть командную строку в редакторе кода Visual Studio Code и открыть терминал Как итог нам необходимо два поля ввода, в которые пользователь может вводить цифры и пока базовые 4 операции это умножение деление вычитание и сложение.

```
<template>
   <div>
       <div class="display">
           <input v-model="operand1" />
           <input v-model="operand2" />
       </div>
       <div class="keyboard">
           <button>+</button>
           <button>-</button>
           <button>/</button>
           <button>*</button>
       </div>
   </div>
</template>
```

```
<script>
export default {
   name: 'CalcTemp',
   data() {
       return {
           operand1: 0,
           operand2: 0,
       }
   }
}
</script>
```

Добавим еще одно значение - в наш data объект - поле result в котором будет храниться результат выполнения математического действий. Сначала добавим свойство в data, и потом добавим вывод его в шаблон:

```
<div class="display">
          <input v-model.number="operand1" />
          <input v-model.number="operand2" />
          = {{ result }}
</div>
```

```
   data() {
       return {
           operand1: 0,
           operand2: 0,
           result: 0,
       }
   }
```

Осталось добавить действия

```
<div class="keyboard">
   <button @click="result = operand1 + operand2">+</button>
   <button @click="result = operand1 - operand2">-</button>
```

```
    <button @click="result = operand1 / operand2">/</button>
    <button @click="result = operand1 * operand2">*</button>
</div>
```

Хорошей практикой считается разделять представление и логику приложения. Не даром у нас компонент разделен на 3 самостоятельные секции – верстка, скрипты и стили. Поэтому, логичным действием будет размещать операции, которые производит компонент, именно в блоке <script>, а не в шаблоне. Во Vue для этих нужд есть специальная секция – блок methods. В этом блоке можно описывать любые функции, которые выполняют бизнес-логику компонента, и которые будут доступны для вызова внутри нашего шаблона.

```
<template>
 ...
  <button @click="add">+</button>
  <button @click="substract">-</button>
  <button @click="divide">/</button>
  <button @click="multiply">*</button>
 ...
</template>
```

В скрипте добавляем нужные методы

```
methods: {
    add() {
        this.result = this.operand1 + this.operand2
    },
    substract() {
        this.result = this.operand1 - this.operand2
    },
    divide() {
        this.result = this.operand1 / this.operand2
    },
    multiply() {
        this.result = this.operand1 * this.operand2
    },
},
```

Обратите внимание, что мы не ставим круглые скобочки у методов, которые хотим вызвать при возникновении события. Однако, может возникнуть ситуация, когда при вызове функции нам необходимо передать в нее какой-нибудь параметр. Например: в нашем распоряжении находятся не 4 отдельные функции, каждая из которых представляет свою арифметическую операцию, а лишь одна функция-агрегатор. Такая функция может принимать в себя только знак арифметической операции, и на основании этого знака высчитывать результат. Давайте напишем эту функцию и назовем ее calculate.

```
calculate(operation = '+') {
            switch (operation) {
                case '+':
                    this.add()
                    break;
                case '-':
                    this.substract()
                    break;
                case '*':
                    this.multiply()
                    break;
                case '/':
                    this.divide()
                    break;
            }
},
```

Теперь в нашем шаблоне мы можем использовать лишь одну функцию calculate. Знак операции мы можем передать совершенно обычным способом. В отличие от других фреймворков (например, в отличие от React), шаблон во Vue не будет запускать функцию, в которую передается параметр, до наступления соответствующего события:

```
        <div class="keyboard">
            <button @click="calculate('+')">+</button>
            <button @click="calculate('-')">-</button>
            <button @click="calculate('/')">/</button>
```

```
            <button @click="calculate('*')">*</button>
        </div>
```

Директивы

Как мы уже ранее говорили, при разработке приложения следует разделять логику и отображение. Именно поэтому мы перенесли работу с высчитыванием результата из шаблона в отдельные методы. Однако, в то же время, Vue нам предоставляет специальный расширенный функционал в шаблонах, с помощью которого мы можем управлять нашей версткой “прямо на месте”. Делается это с помощью так называемых директив.

Директивы - это специальные атрибуты у элементов в шаблоне, которые управляют отображением этих самых элементов, или наделяют их какой-то логикой. Мы уже встречались с некоторыми директивами, например v-model. Все директивы начинаются с префикса ‘v-’, благодаря которыми их нельзя спутать с обычными атрибутами.

Директивы работают на основании данных компонента. То есть, они принимают в себя какое-то JavaScript выражение (например данные из блока data) и на основании этих данных выполняют какую-то работу с элементами шаблона. Главным удобством при работе с директивами является их реактивность: если данные, которые передаются в директиву обновятся, то и шаблон обновит свое отображение.
Есть целый набор директив, работающих “из коробки” и являющихся частью API Vue, давайте рассмотрим некоторые, самые часто используемые из них.

Директивы условного рендеринга.

Наиболее частой проблемой при создании шаблона является условный рендеринг элементов. В зависимости от значения тех или иных данных мы хотим либо отобразить какой-то элемент, либо скрыть его.

Например, вернемся к нашему калькулятору. Мы знаем, что в JavaScript - язык особенный, он позволяет пользователю без проблем делить на 0. Результатом такой операции будет специальное числовое значение - Infinity. Однако, обычно калькуляторы показывают в таком случае ошибку. Мы тоже хотим себе такой

функционал - если произошло какое-то нехорошие действие, то показать пользователю соответствующее сообщение. Но, если ошибки нет, то и сообщение не стоит показывать. Как это можно сделать?

Для начала, давайте заведем себе в блоке data новое свойство - error. Именно в нем мы будем хранить текст ошибки, и именно на основании этого свойства мы будем принимать решение - показывать пользователю сообщение с ошибкой или нет.

```
data() {
    return {
        operand1: 0,
        operand2: 0,
        result: 0,
        error: '',
    }
},
```

Теперь, в методе деления, при делении на 0, давайте будем заполнять это свойство соответствующим сообщением:

```
    divide() {
        const { operand1, operand2 } = this
        if (operand2 === 0) {
            this.error = 'Делить на 0 нельзя!'
        } else {
            this.result = operand1 / operand2
        }
    },
```

Также не стоит забывать об обнулении ошибки. Лучше всего это сделать перед началом операции. Сделаем это перед блоком switch в методе calculate:

```
    calculate(operation = '+') {
        this.error = ''
        switch (operation) {
            . . .
```

Теперь осталось лишь показывать ошибку в шаблоне, лишь в том случае, если она не пустая. На помощь нам придут директивы условного рендеринга - v-if и v-show.

```
<div v-show="error">Ошибка! {{ error }}</div>
```

Директива v-if очень похожа на одноименный операнд в JavaScript (да и в любом другом языке программирования). Если условие выполнено, то выполняем (рендерим) блок, который идет следом за условием. Также, как и в языках программирования, логику условия можно расширить, с помощью директив v-else, а также v-else-if.

Давайте представим такую ситуацию. Продакт менеджер нашего калькулятора провел исследование и выяснил, что в зависимости от результата выражения, нам просто необходимо показать пользователю определенные сообщения, а именно:

- Если результат выражения менее 0, сообщение “Получилось отрицательное число”
- Если результат выражения лежит в пределах от 0 до 100, сообщение “Результат в первой сотне”
- Если результат больше 100, вывести “Получилось слишком большое число”

Как с помощью изученных директив v-if/v-else-if/v-else можно удовлетворить требованию продукта и вывести эти сообщения? Очень просто:

```
<div class="strange-message">
   <template v-if="result < 0">Получилось отрицательное
число</template>
   <template v-else-if="result < 100">Результат в первой
сотне</template>
   <template v-else>Получилось слишком большое число</template>
</div>
```

Здесь важно отметить, что если мы используем эти дополнительные директивы, то обязаны использовать их строго в представленном выше порядке. Директива v-if должна быть первой в списке (мы же не можем поставить “иначе” не задав начальное условие, верно?), а директива v-else должна заканчивать перечисление условий. Между v-if и v-else могут располагаться только v-else-if директории.

Отрисовка списков

С этой частью мы уже знакомы, так что тут можно смело переводить код с использованием v-for

```
<div class="keyboard">
      <button  v-for="operand  in  operands"  v-bind:key="operand"
@click="calculate(operand)">
       {{ operand }}
   </button>
</div>
```

```
        return {
            operand1: 0,
            operand2: 0,
            result: 0,
            error: '',
            operands: ['+', '-', '/', '*'],
        }
```

Привязка значений атрибутов

Ранее мы уже встретились с директивой v-bind, то давайте рассмотрим различные вариации ее применения.

```
           <button v-for="operand in operands"
v-bind:key="operand"
v-bind:title="operand"
@click="calculate(operand)">
                {{ operand }}
           </button>
```

Теперь, если мы откроем инспектор элементов и найдем в нем наши кнопки, то увидим у них атрибут title со своими операндами:

```
<div class="keyboard"> == $0
  <button title="+"> + </button>
  <button title="-"> - </button>
  <button title="/"> / </button>
  <button title="*"> * </button>
</div>
```

Теперь, если навести курсором мышки на кнопку, у нее появится тултип с оператор.
Как и со всеми остальными директивами, привязка обладает свойствами реактивности, то есть, при изменении данных, передаваемых в директиву v-bind, шаблон будет перерендерен. Давайте продемонстрируем эту особенность: на основании наличия данных в полях для ввода мы будем делать кнопки активными или задизейбленными. Если хотя бы одно из полей для ввода будет пустым, пользователь не сможет нажать кнопку для выполнения арифметической операции.

```
          <button v-for="operand in operands"
v-bind:key="operand"
v-bind:title="operand"
v-bind:disabled="!operand1 || !operand2"
@click="calculate(operand)">
              {{ operand }}
          </button>
```

Теперь, если нам не хватает данных для выполнения операции, кнопка будет становиться задизейбленной.
Мы уже с вами не раз упоминали такой термин, как реактивность. Давайте попробуем разобраться разобраться, что же это такое?

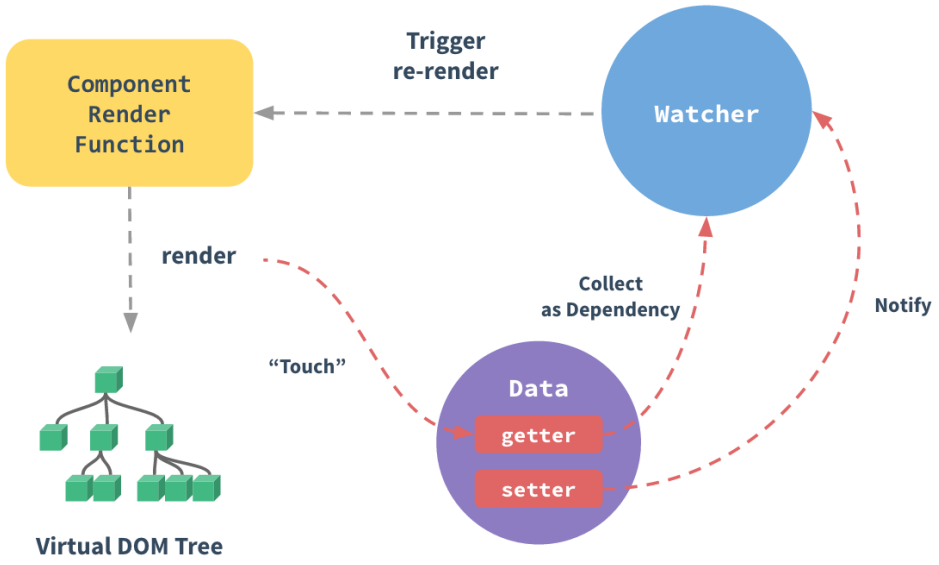
Реактивность во Vue

Если не вдаваться в техническое определение, то понятие реактивности объединяет в себя процесс изменения состояние объекта, с последующим изменением состояний всех зависимых от него сущностей. Среди зависимых сущностей могут быть как объекты, которые используются в качестве параметров другими компонентами, директивами (наподобие директив v-if/v-show/v-bind и прочих), так и целые блоки шаблона, которые рендерятся на основании значения исходного объекта.

Понятная сходу реализация реактивности во фреймворке - одна из его важных особенностей которая позволяет быстро начать его использовать.

Мы говорим о реактивности как о двусторонней связи того, что отображается на странице, и данных, которые описываются в JS коде - эту часть часто называют моделью данных.

Модели представляют собой простые JavaScript-объекты. По мере их изменения, обновляется и представление данных, благодаря чему управление состоянием приложения становится простым и очевидным. Верно и обратное - изменение значений в полях ввода, клики по кнопкам и прочие события вызывают методы из компонента, которые в свою очередь изменяют значение модели, то есть самих данных.



Однако, при работе с реактивностью, стоит внимательно следить за обновлением данных в модели данных. Когда мы работаем с примитивными типами данных, такими как строка, число или логическое значение - проблем никаких нет. Однако, когда мы начинаем работать с объектами, необходимо помнить, что Vue отслеживает лишь изменения тех свойств, которые были в объекте изначально. Если в процессе жизни приложения в объект с данными добавляются новые свойства, реактивность не сработает.

Рассмотрим данную особенность на примере. Предположим, что продукт менеджер нашего калькулятора придумал функционал, который необходимо реализовать, а именно - сохранение ранее проведенных операций. Давайте сделаем это.

Первым делом необходимо завести место, где мы будем хранить логи:

```
  data() {
```

```
        return {
            operand1: 0,
            operand2: 0,
            result: 0,
            error: '',
            operands: ['+', '-', '/', '*'],
            logs: {}, // здесь будем хранить наши логи
        }
    },
```

В качестве ключа объекта logs будет выступать время, в которое была сделана операция, в качестве значения - выражение и результат. Заполним этот объект в методе calculate:

```
        calculate(operation = '+') {
            this.error = ''
            switch (operation) {
. . .
            }
            this.logs[Date.now()] =
`${this.operand1}${operation}${this.operand2}=${this.result}`
```

Осталось только вывести наши логи. Сделаем это с помощью уже знакомой нам директивы v-for, которая отлично умеет работать с объектами:

```
<div class="logs">
    <div v-for="(log, id) in logs" v-bind:key="id">{{ log }}</div>
</div>
```

В id будет храниться ключ объекта, то есть время, в которое была сделана операция. Время хранится в миллисекундах, поэтому можно принять за истину, что ключи у нас будут уникальными. Раз ключи уникальные, то воспользуемся ими, чтобы заполнить атрибут key.

Если мы сейчас запустим наше приложение и начнем нажимать на кнопки операций, то мы увидим, что никакие логи у нас не показываются. И не будут показываться до тех пор, пока у нас не изменится результат нашего выражения (то есть не будет перерендерен шаблон). Это как раз и происходит вследствие того, что мы добавляем свойства, которых ранее не было в объекте logs. Добавление новых свойств Vue не видит, и не пытается перерендерить шаблон, хотя он и зависит от

этих данных. Однако, способ способ заставить реактивность запуститься все же есть.

Vue.set

Метод Vue.set -позволяет сообщить внутренней системе фреймворка что мы хотим добавить или заменить какие-то и свойства существующего объекта или массива.
Метод vue.set() используется для добавления новых свойств в реактивный объект Vue.js. Объекты, которые были объявлены с помощью data() становятся реактивными, то есть, когда свойства этих объектов изменяются, Vue.js автоматически обновляет соответствующие элементы в DOM. Однако, если вы попытаетесь добавить новое свойство в объект Vue.js, которое не было определено изначально в data(), Vue.js не будет знать об этом свойстве и не сможет автоматически отслеживать его изменения.

Записывается синтаксис так - Vue.set(object, propertyName, value). Метод можно вызвать на глобальном экземпляре -

```
import Vue from 'vue'
...
Vue.set(object, propertyName, value)
```

или так - внутри методов компонента

```
this.$set(object, propertyName, value)
```

Давайте применим данный способ для обновления наших логов:

```
            switch (operation) {
. . .
            }
            const key = Date.now()
            const value =
`${this.operand1}${operation}${this.operand2}=${this.result}`
            Vue.set(this.logs, key, value)
```

Не хватает только подключения экземпляра vue

```
<script>
import Vue from 'vue'
```

Теперь, если мы запустим наше приложение, то на каждую операцию у нас будет появляться запись в логе, даже, если результат операции не поменяется! Данным методом мы еще не раз будем пользоваться

Что дальше?

Мы уже сделали много полезного функционала в нашем калькуляторе. Но можем сделать еще больше. Давайте представим ситуацию: аналитики провели опросы среди пользователей и выяснили, что помимо операций над простыми числами, необходимо проводить операции над числами в последовательности Фибоначчи.

Вроде бы сделать не сложно: при каждом вызове метода calculate вычислять от операндов значения и выводить результат. Давайте сделаем это. Первым делом напишем метод, который будет вычислять число в последовательности Фибоначчи:

```
fib(n) {
   return n <= 1 ? n : this.fib(n - 1) + this.fib(n - 2);
},
```

Теперь надо вычислять результат от вычисленных значений. Сохранять его будем в новую переменную fibResult.

```
add () {
 this.result = this.operand1 + this.operand2
 this.fibResult = this.fib(this.operand1) +
this.fib(this.operand2)
},
substract () {
 this.result = this.operand1 - this.operand2
 this.fibResult = this.fib(this.operand1) -
this.fib(this.operand2)
},
// и так далее
```

Выведем в шаблон:

Наша программа работает и работает правильно. Однако, выглядит немного громоздко, да и при тестировании мы можем заметить, что при изменении лишь одного операнда, у нас заново пересчитываются наши значения последовательности Фибоначчи. Учитывая, что эта операция не самая быстрая в плане производительности, мы теряем много времени. Было бы здорово, если бы мы могли высчитывать значения лишь тогда, когда у нас меняются входные

параметры функции. Если изменяется только операнд 1, то не пересчитывать значение операнда 2 и наоборот. Во Vue у нас есть такая возможность.

Вычисляемые свойства - computed

Вычисляемые свойства - специальные методы во Vue, которые обязаны возвращать результат. Также, в отличие от обычных методов, такие функции будут выполняться лишь один раз, после чего они кэшируют результат и при последующих обращениях отдают уже готовое значение (особенно удобно это при долгих вычислениях, как, например, в нашем случае с числами Фибоначчи). Если при выполнении используются реактивные данные, и эти данные были изменены, тогда произойдет перерасчет значения. Давайте применим данную практику к нашему функционалу. Вычисляемые свойства описываются в специальной секции - computed.

```
   computed: {
        fibb1() {
            return this.fib(this.operand1)
        },
        fibb2() {
            return this.fib(this.operand2)
        },
   },
```

Используем новые данные в наших операциях. К вычисляем свойствам обращаются именно как к свойствам, а не как к методом, то есть без использования круглых скобок при вызове:

```
add () {
 this.result = this.operand1 + this.operand2
 this.fibResult = this.fibb1 + this.fibb2
},
substract () {
 this.result = this.operand1 - this.operand2
 this.fibResult = this.fibb1 - this.fibb2
},
```

Если мы протестируем наше приложение сейчас, то сможем заметить прирост скорости, в случаях, когда мы изменяем лишь один из операндов. Это может

означать, что кэширование работает успешно, и у нас не выполняются лишние операции. Оптимизация прошла успешно, и наш менеджер остался нами доволен! Поэтому, работу над калькулятором мы завершаем и готовимся к новым интересным проектам, которые ждут нас в следующих уроках :)

Итоги урока

Мы подробно разобрали установку и запуск Vue CLI, разобрали возможнжости редактирования и конечно же рассмотрели все созданные элементы, не стоит бояться большого количества папко или файлов, ведь вам не нужно будет работать со всеми элементами сразу, плюс стандартный шаблон содержит в себе много дополнительного кода, который на старте можно смело удалить и тогда Vue CLI станет для вас самым оптимальным решением и уже использовать стандартное подключение через CDN точно не потребуется. Теперь остается только разбить проект на блоки и создать все необходимые компоненты, а как это сделать мы узнаем на следующем уроке

Используемая литература

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