GeekBrains

Методы и работа с данными

Vue.js

Оглавление

Введение

В прошлом уроке мы успели познакомиться с фреймворком Vue.js научились запускать первый код и создавать самые простые конструкции, важной особенностью конечно являются события, которые мы обрабатываем на странице.

На этой лекции вы найдете ответы на такие вопросы как / узнаете:

- Встроенные директивы
- Что такое методы и как с ними работать
- Работа с данными
- Добавление стилизации

Директивы

Еще одной важной концепцией Vue являются **директивы**. Для описания шаблонов мы используем html с дополнительными конструкциями, которые помогают связывать представление с данными. Мы уже говорили об {{ интерполяциях }}, пришло время поговорить еще об одном способе связывать наши данные с шаблоном.

Директива - это специальный атрибут html элемента или компонента с приставкой v-, который “привязывает” изменение этого элемента к какому-то js выражению.
Рассмотрим одну из популярных встроенных директив Vue.

Директива v-on

Прикрепляет к элементу подписчик события. Тип события указывается в параметре после тире, например:

```
v-on:click
v-on:input
v-on:focus
v-on:mouseover
v-on:keyup
```

Мы уже с вами работали с click событием, так что все остальные директивы будет узнать еще проще.
Давайте посмотрим несколько интересных вариантов использования v-on:

```
<!-- 1: inline-выражение -->
<button v-on:click="c = a + b"></button>

<!-- 2: вызов метода обработчика с параметрами -->
<button v-on:click="doThat('hello', $event)"></button>

<!-- 3: вызов метода обработчика без параметров -->
<button v-on:click="doThis"></button>
```

У обычного элемента можно подписаться только на нативные события DOM. У элемента компонента можно подписаться на пользовательские события, вызываемые этим дочерним компонентом - это мы разберем в следующих уроках.

Напоминаю, что все события v-on: можно сократить до формата @ и как итог получить @input

Директива v-bind

Данная директива позволяет “привязать” значение атрибута. Простыми словами, то если мы хотим в html-атрибут элемента передать значение из данных data, то v-bind нам в этом может помочь.
Давайте привяжем значение title внутри data использую v-bind

```
<button v-bind:id="id" v-bind:title="title">Кнопка</button>
```

script.js

```
new Vue({
    el: '#app',
    data: {
        id: '1',
        title: 'button'
    }
})
```

С помощью данной директивы, можно также привязывать стили для атрибутов, через

`:class="className"`

Как мы можем заметить сокращенная запись v-bind:id будет просто :id

При использовании с булевыми атрибутами (когда их наличие уже означает true) v-bind обладает некоторыми особенностями. Давайте рассмотрим пример, в котором будем делать кнопку не активной

В этом примере:

```
<div id="app">
    <button :disabled="isDisabled">Кнопка</button>
</div>
```

```
<script>
    new Vue({
        el: '#app',
        data: {
            isDisabled: true
        }
    })
</script>
```

Директива v-model

V-model помогает связать элемент для ввода с какой-то переменной в наших данных. Причем сделать это в двух направлениях: таким образом, что при изменении данных в поле ввода будут меняться данные в объекте data, и наоборот.

Нам необходимо работать с полями ввода, как понять какие значения мы в них помещаем, тут к нам на помощь и приходит v-model, простыми словами мы моделируем данные внутри полей ввода и теперь можем их записывать в переменные, смотрим на реализацию ниже, где мы будем моделировать operand1 и operand2

```
<div id="app">
    <div class="display">
        <input v-model="operand1">
        <input v-model="operand2">
        = {{ result }}
    </div>
</div>
<script>
    new Vue({
        el: '#app',
        data: {
            operand1: 0,
            operand2: 0,
            result: 0,
        }
    })
</script>
```

Модификаторы в директивах

У каждой директивы могут быть свои модификаторы. Они позволяют дополнительно сконфигурировать поведение директивы, выполняя какие-то рутинные операции.
Для примера у v-model есть такие модификаторы:

- .trim - автоматически обрезать пробелы в начале и в конце строки
- .lazy - v-model синхронизирует ввод с данными по событию input, чтобы использовать для синхронизации после события change
- .number - для автоматического приведения введённого пользователем к Number

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

Теперь, в свойства operand1 и operand2 при вводе данных будут записываться числа, а не текст.

Отрисовка списков

Второй из наиболее часто встречающихся проблем в составлении шаблонов, является проблема отображения данных из массива. Для рендеринга подобных коллекций, Vue предоставляет разработчикам специальную директиву v-for.
У данной директивы особый синтаксис входного выражения, который может напоминать хорошо известный нам метод итерации по массиву через операнд for in. Выглядит запись следующим образом: item in myCollection, где item - элемент массива, а myCollection соответственно сам массив.

```
<div id="app">
    <div v-for="item in myCollection">
        {{ item }}
    </div>
</div>
<script>
    new Vue({
        el: '#app',
        data: {
            myCollection: [1, 2, 3, 4, 5, 6, 7]
        }
```

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

При переборе элементов с помощью дополнительного параметра в v-for мы можем получить индекс элемента в массиве, используя следующий синтаксис:

```
<div v-for="(item, index) in myCollection">
    {{ index }} - {{ item }}
</div>
```

Где item - это текущий элемент в массиве myCollection, а index - индекс этого элемента в массиве.

Методы

Мы научились работать с блоком data, сейчас в него сохраняются данные из полей ввода, а также в этом блоке предусмотрен результат, который нам и предстоит узнать.

Давайте продумаем логику, как необходимо вычислить результат, конечно нам необходимо предоставить пользователю выбор между четырех операций, умножение деление, сложение и вычитание, для реализации таких действий нам как раз потребуются методы.
Первый вариант который возможен для подсчета результата это конечно уже знакомая нам директива v-on:click внутри которой мы можем посчитать результат, код будет выглядеть таким образом

```
<div class="display">
    <input v-model.number="operand1">
    <input v-model.number="operand2">
    = {{ result }}
</div>
<div class="keyboard">
    <button v-on:click="result = operand1 + operand2">+</button>
    <button v-on:click="result = operand1 - operand2">-</button>
    <button @click="result = operand1 / operand2">/</button>
    <button @click="result = operand1 * operand2">*</button>
</div>
```

В первых двух кнопках используется старый синтаксис, а в умножении и делении уже в сокращенном внешнем виде, тут нет какой-то правильности, вы можете выбирать и тот и тот вариант, но чаще используется @

Хорошей практикой считается разделять представление и логику приложения. Поэтому, логичным действием будет размещать операции, которые производит компонент, именно в блоке <script>. Во Vue для этих нужд есть специальная секция – блок methods. В этом блоке можно описывать любые функции, которые выполняют бизнес-логику.
Давайте добавим еще одну кнопку

```
<button @click="add">add</button>
```

и теперь нужно добавить сам метод

```
new Vue({
   el: '#app',
   data: {
       operand1: 0,
       operand2: 0,
       result: 0,
   },
   methods: {
       add() {
           console.log('Add operation!')
       }
   },
})
```

Теперь при клике на данную кнопку (add) вызывается метод add, который выводит в консоль текст “Add operation!”

Давайте перенесем всю существующую логику выполнения арифметических операций из шаблона в блок methods. Как будет выглядеть наш блок со скриптом:

```
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
    },
},
```

Вроде бы ничего сложного нет, мы просто скопировали те операции, которые находились в обработке событий кнопок и перенесли их в соответствующие функции. Однако, если более внимательно посмотреть на код, который мы разместили в новых методах, мы можем обратить внимание, что обращение к данным из блока data теперь происходит через ключевое слово this. Это действительно так – по своей сути стандартная работа с объектом. Чтобы обратиться к свойствам и методам объекта, внутри данного объекта мы обязаны использовать ключевое слово this.

Отлично! У нас появился новый функционал, давайте воспользуемся им – вызовем написанные нами функции в нашем html.

```
<div class="keyboard">
    <button @click="add">+</button>
    <button @click="substract">-</button>
    <button @click="divide">/</button>
    <button @click="multiply">*</button>
</div>
```

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.

Вычисляемые свойства

Теперь давайте немного отвлечемся от нашего калькулятора и подумаем над проблемой нашего method, ведь он считает наши значения во всех данных внутри метода, даже если мы их не используем, получается что мы расходуем ресурсы компьютера в пустую, именно для этого придумали вычисляемые свойства

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. Если при выполнении используются реактивные данные, и эти данные были изменены, тогда произойдет перерасчет значения.

Конечно тут без примера не разобраться

```
<div id="example">
   {{ message.split('').reverse().join('') }}
</div>
```

Такой шаблон уже не выглядит простым и декларативным. С первого взгляда и не скажешь, что он всего лишь отображает message задом наперёд. Ситуация станет ещё хуже, если эту логику потребуется использовать в нескольких местах шаблона.

На помощь здесь приходят вычисляемые свойства.

```
<div id="example">
   <p>Изначальное сообщение: «{{ message }}»</p>
   <p>Сообщение задом наперёд: «{{ reversedMessage }}»</p>
</div>
<script>
   new Vue({
       el: '#example',
       data: {
           message: 'Привет'
```

```
        },
        computed: {
            // геттер вычисляемого значения
            reversedMessage: function () {
                // `this` указывает на экземпляр vm
                return this.message.split('').reverse().join('')
            }
        }
    })
</script>
```

Мы определили вычисляемое свойство reversedMessage. Написанная нами функция будет использоваться как геттер свойства vm.reversedMessage:

```
console.log(vm.reversedMessage) // => 'тевирП'
vm.message = 'Пока'
console.log(vm.reversedMessage) // => 'акоП'
```

Вы можете открыть консоль и поиграть с примером самостоятельно. Значение vm.reversedMessage всегда зависит от значения vm.message.

В шаблонах можно обращаться к вычисляемым свойствам как и к обычным. Vue знает, что vm.reversedMessage зависит от vm.message, поэтому при обновлении vm.message обновятся и все зависящие от него элементы, в нашем случае обновится vm.reversedMessage. Самое важное — эту зависимость теперь мы указали декларативно: геттер вычисляемого свойства не имеет побочных эффектов, что упрощает понимание кода и его тестирование.

Можно заметить, что такого же результата можно достичь и с помощью метода:

```
<div id="example">
    <p>Сообщение задом наперёд: «{{ reverseMessage() }}»</p>
</div>
<script>
    new Vue({
        el: '#example',
```

```
        data: {
            message: 'Привет'
        },
        methods: {
            reverseMessage: function () {
                return this.message.split('').reverse().join('')
            }
        },
    })
</script>
```

Вместо вычисляемого свойства, можно использовать ту же самую функцию в качестве метода. С точки зрения конечного результата, оба подхода делают одно и то же. Но есть важное отличие: **вычисляемые свойства кэшируются, основываясь на своих реактивных зависимостях**. Вычисляемое свойство пересчитывается лишь тогда, когда изменится одна из его реактивных зависимостей. Поэтому, пока message остаётся неизменным, многократное обращение к reversedMessage будет каждый раз возвращать единожды вычисленное значение, не запуская функцию вновь.

Используемая литература

1. https://ru.vuejs.org/