EBook Gratuito

APPRENDIMENTO Vue.js

Free unaffiliated eBook created from
Stack Overflow contributors.

#vue.js

Sommario

Di

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: vue-js

It is an unofficial and free Vue.js ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Vue.js.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capitolo 1: Iniziare con Vue.js

Osservazioni

`Vue.js` è un framework front-end in crescita rapida per `JavaScript` , ispirato ad `Angular.js` , `Reactive.js` e `Rivets.js` che offre una progettazione semplicistica dell'interfaccia utente, manipolazione e reattività profonda.

È descritto come un framework modellato `MVVM` , `Model-View View-Model` , che si basa sul concetto di *associazione dei* dati di *binding* a componenti e viste. È incredibilmente veloce, supera la velocità di altri framework `JS` alto livello e è molto intuitivo per una facile integrazione e prototipazione.

Versioni

Versione	Data di rilascio
2.4.1	2017/07/13
2.3.4	2017/06/08
2.3.3	2017/05/09
2.2.6	2017/03/26
2.0.0	2016/10/02
1.0.26	2016/06/28
1.0.0	2015/10/26
0.12.0	2015/06/12
0.11.0	2014/11/06

Examples

"Ciao mondo!" Programma

Per iniziare a utilizzare Vue.js , assicurati di avere il file di script incluso nel codice HTML. Ad esempio, aggiungi quanto segue al tuo codice HTML.

```
<script src="https://npmcdn.com/vue/dist/vue.js"></script>
```

Semplice esempio

Modello HTML

```
<div id="app">
  {{ message }}
</div>
```

JavaScript

```
new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue.js!'
  }
})
```

Guarda una demo dal vivo di questo esempio.

Potresti anche voler controllare l'esempio "Hello World" creato da Vue.js.

Hello World in Vue 2 (The JSX way)

JSX non è pensato per essere interpretato dal browser. Deve essere prima trasposto in Javascript standard. Per usare JSX devi installare il plugin per `babel-plugin-transform-vue-JSX`

Esegui il comando qui sotto:

```
npm install babel-plugin-syntax-jsx babel-plugin-transform-vue-jsx babel-helper-vue-jsx-merge-props --save-dev
```

e aggiungilo al tuo `.babelrc` questo modo:

```
{
  "presets": ["es2015"],
  "plugins": ["transform-vue-jsx"]
}
```

Codice di esempio con VUE JSX:

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

new Vue({
  el: '#app',
  methods: {
    handleClick () {
      alert('Hello!')
```

```
    }
  },
  render (h) {
    return (
      <div>
        <h1 on-click={this.handleClick}>Hello from JSX</h1>
        <p> Hello World </p>
      </div>
    )
  }
})
```

Usando JSX puoi scrivere concise strutture HTML / XML nello stesso file di cui scrivi il codice JavaScript.

Congratulazioni, hai fatto :)

Gestire l'input dell'utente

VueJS può essere utilizzato per gestire facilmente anche l'input dell'utente, e l'associazione bidirezionale con v-model rende davvero facile cambiare i dati facilmente.

HTML:

```
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<div id="app">
    {{message}}
<input v-model="message">
</div>
```

JS:

```
new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue.js!'
  }
})
```

È molto facile eseguire un binding a due vie in VueJS usando la direttiva `v-model`.

Guarda un esempio dal vivo qui.

Leggi Iniziare con Vue.js online: https://riptutorial.com/it/vue-js/topic/1057/iniziare-con-vue-js

Capitolo 2: Associazione dati

Examples

Testo

La forma più semplice di associazione dei dati è l'interpolazione del testo usando la sintassi "Moustache" (doppie parentesi graffe):

```
<span>Message: {{ msg }}</span>
```

Il tag dei baffi verrà sostituito con il valore della proprietà `msg` sull'oggetto dati corrispondente. Verrà inoltre aggiornato ogni volta che la proprietà `msg` dell'oggetto dati cambia.

È inoltre possibile eseguire interpolazioni una tantum che non si aggiornano sulla modifica dei dati:

```
<span>This will never change: {{* msg }}</span>
```

HTML grezzo

I baffi doppi interpretano i dati come testo semplice, non HTML. Per generare un vero codice HTML, dovrai utilizzare i baffi tripli:

```
<div>{{{ raw_html }}}</div>
```

I contenuti sono inseriti come semplici HTML - i collegamenti dei dati vengono ignorati. Se hai bisogno di riutilizzare i pezzi del modello, dovresti usare partial.

attributi

I baffi possono essere utilizzati anche all'interno degli attributi HTML:

```
<div id="item-{{ id }}"></div>
```

Tieni presente che le interpolazioni degli attributi non sono consentite nelle direttive Vue.js e negli attributi speciali. Non preoccuparti, Vue.js solleva avvisi per te quando i baffi vengono utilizzati in posti sbagliati.

filtri

Vue.js ti consente di aggiungere "filtri" opzionali alla fine di un'espressione, indicati dal simbolo "pipe":

```
{{ message | capitalize }}
```

Qui stiamo "convogliando" il valore dell'espressione del `message` attraverso il filtro in `capitalize` incorporato, che in realtà è solo una funzione JavaScript che restituisce il valore in maiuscolo. Vue.js fornisce una serie di filtri integrati e parleremo di come scrivere i tuoi filtri in un secondo momento.

Nota che la sintassi della pipe non fa parte della sintassi JavaScript, quindi non puoi mischiare i filtri all'interno delle espressioni; puoi solo aggiungerli alla fine di un'espressione.

I filtri possono essere concatenati:

```
{{ message | filterA | filterB }}
```

I filtri possono anche prendere argomenti:

```
{{ message | filterA 'arg1' arg2 }}
```

La funzione filtro riceve sempre il valore dell'espressione come primo argomento. Gli argomenti citati sono interpretati come una stringa semplice, mentre quelli non quotati saranno valutati come espressioni. Qui, la stringa semplice `'arg1'` verrà passata nel filtro come secondo argomento, e il valore dell'espressione `arg2` verrà valutato e passato come terzo argomento.

Leggi Associazione dati online: https://riptutorial.com/it/vue-js/topic/1213/associazione-dati

Capitolo 3: Bus degli eventi

introduzione

I bus di eventi sono un modo utile di comunicare tra componenti che non sono direttamente correlati, cioè non hanno alcuna relazione genitore-figlio.

È solo un'istanza `vue` vuota, che può essere utilizzata per `$emit` eventi o ascoltare `$on` eventi citati.

Sintassi

1. export default new Vue ()

Osservazioni

Usa vuex se la tua applicazione ha molti componenti che richiedono i dati l'uno dell'altro.

Examples

EventBus

```
// setup an event bus, do it in a separate js file
var bus = new Vue()

// imagine a component where you require to pass on a data property
// or a computed property or a method!

Vue.component('card', {
  template: `<div class='card'>
    Name:
      <div class='margin-5'>
          <input v-model='name'>
    </div>
    <div class='margin-5'>
      <button @click='submit'>Save</button>
    </div>
  </div>`,
  data() {
    return {
      name: null
    }
  },
  methods: {
    submit() {
      bus.$emit('name-set', this.name)
    }
  }
})

// In another component that requires the emitted data.
```

```
var data = {
  message: 'Hello Vue.js!'
}

var demo = new Vue({
  el: '#demo',
  data: data,
  created() {
    console.log(bus)
    bus.$on('name-set', (name) => {
      this.message = name
    })
  }
})
```

Leggi Bus degli eventi online: https://riptutorial.com/it/vue-js/topic/9498/bus-degli-eventi

Capitolo 4: componenti

Osservazioni

In Component (s):

oggetti di scena è una serie di stringhe letterali o riferimenti a oggetti usati per passare dati dal componente principale. Può anche essere in forma di oggetto quando si desidera avere un controllo più fine come specificare i valori predefiniti, il tipo di dati accettati, se è richiesto o facoltativo

i dati devono essere una funzione che restituisce un oggetto invece di un oggetto semplice. È così perché richiediamo a ciascuna istanza del componente di avere i propri dati per finalità di riusabilità.

eventi è un oggetto che contiene listener per eventi a cui il componente può rispondere in base a cambiamenti comportamentali

metodi oggetto contenente funzioni che definiscono il comportamento associato al componente

le proprietà **calcolate** sono come osservatori o osservabili, ogni volta che una dipendenza cambia le proprietà vengono ricalcolate automaticamente e le modifiche si riflettono nel DOM immediatamente se DOM utilizza qualsiasi proprietà calcolata

ready è il gancio del ciclo di vita di un'istanza Vue

Examples

Ambito del componente (non globale)

dimostrazione

HTML

```
<script type="x-template" id="form-template">
    <label>{{inputLabel}} :</label>
  <input type="text" v-model="name" />
</script>

<div id="app">
  <h2>{{appName}}</h2>
  <form-component title="This is a form" v-bind:name="userName"></form-component>
</div>
```

JS

```
// Describe the form component
// Note: props is an array of attribute your component can take in entry.
// Note: With props you can pass static data('title') or dynamic data('userName').
// Note: When modifying 'name' property, you won't modify the parent variable, it is only
descendent.
// Note: On a component, 'data' has to be a function that returns the data.
var formComponent = {
  template: '#form-template',
  props: ['title', 'name'],
  data: function() {
    return {
      inputLabel: 'Name'
    }
  }
};

// This vue has a private component, it is only available on its scope.
// Note: It would work the same if the vue was a component.
// Note: You can build a tree of components, but you have to start the root with a 'new
Vue()'.
var vue = new Vue({
  el: '#app',
  data: {
    appName: 'Component Demo',
    userName: 'John Doe'
  },
  components: {
    'form-component': formComponent
  }
});
```

Quali sono i componenti e come definire i componenti?

I componenti in Vue sono come i widget. Ci permettono di scrivere elementi personalizzati riutilizzabili con il comportamento desiderato.

Non sono altro che oggetti che possono contenere una o tutte le opzioni che la root o qualsiasi istanza di Vue può contenere, incluso un template HTML da renderizzare.

I componenti consistono in:

- Markup HTML: il modello del componente
- Stili CSS: come verrà visualizzato il markup HTML
- Codice JavaScript: i dati e il comportamento

Questi possono essere scritti ciascuno in un file separato o come un singolo file con estensione `.vue`. Di seguito sono riportati esempi che mostrano entrambi i modi:

.VUE - come singolo file per il componente

```
<style>
    .hello-world-compoment{
        color:#eeeeee;
        background-color:#555555;
    }
</style>
```

```
<template>
    <div class="hello-world-component">
        <p>{{message}}</p>
        <input @keyup.enter="changeName($event)"/>
    </div>
</template>

<script>
    export default{
        props:[ /* to pass any data from the parent here... */ ],
        events:{ /* event listeners go here */},
        ready(){
            this.name= "John";
        },
        data(){
           return{
              name:''
           }
        },
        computed:{
            message(){
                return "Hello from " + this.name;
            }
        },
        methods:{
            // this could be easily achieved by using v-model on the <input> field, but just
to show a method doing it this way.
            changeName(e){
                this.name = e.target.value;
            }
        }
    }
</script>
```

File separati

hello-world.js - il file JS per l'oggetto componente

```
export default{
    template:require('./hello-world.template.html'),
    props:[ /* to pass any data from the parent here... */ ],
    events:{ /* event listeners go here */ },
    ready(){
        this.name="John";
    },
    data(){
        return{
            name:''
        }
    },
    computed:{
        message(){
            return "Hello World! from " + this.name;
        }
    },
    methods:{
        changeName(e){
            let name = e.target.value;
            this.name = name;
```

```
        }
    }
}
```

ciao-world.template.html

```
<div class="hello-world-component">
    <p>{{message}}</p>
    <input class="form-control input-sm" @keyup.enter="changeName($event)">
</div>
```

ciao-world.css

```
 .hello-world-compoment{
    color:#eeeeee;
    background-color:#555555;
}
```

Questi esempi usano la sintassi es2015, quindi Babel sarà necessario per compilarli in es5 per i browser più vecchi.
Babel con **Browserify + vueify** o **Webpack + vue-loader** sarà richiesto per compilare `hello-world.vue` .

Ora che abbiamo definito il componente `hello-world` , dovremmo registrarlo con Vue.

Questo può essere fatto in due modi:

Registrati come componente globale
Nel file `main.js` (punto di accesso all'app) possiamo registrare qualsiasi componente a livello globale con `Vue.component` :

```
import Vue from 'vue'; // Note that 'vue' in this case is a Node module installed with 'npm
install Vue'
Vue.component('hello-world', require('./hello-world'); // global registeration

new Vue({
    el:'body',

    // Templates can be defined as inline strings, like so:
    template:'<div class="app-container"><hello-world></hello-world></div>'
});
```

Oppure registrarlo localmente all'interno di un componente principale o di un componente root

```
import Vue from 'vue'; // Note that 'vue' in this case is a Node module installed with 'npm
install Vue'
import HelloWorld from './hello-world.js';

new Vue({
    el:'body',
    template:'<div class="app-container"><hello-world></hello-world></div>",
```

```
    components:{HelloWorld}  // local registeration
});
```

I componenti globali possono essere utilizzati ovunque all'interno dell'applicazione Vue.

I componenti locali sono disponibili solo per l'uso nel componente principale con cui sono registrati.

Componente del frammento
Potresti ricevere un errore della console che ti dice che non puoi fare qualcosa perché il tuo è un *componente di frammento* . Per risolvere questo tipo di problema è sufficiente avvolgere il modello del componente all'interno di un singolo tag, come un `<div>` .

Registrazione locale dei componenti

Un componente può essere registrato globalmente o localmente (collegarsi a un altro componente specifico).

```
var Child = Vue.extend({
    // ...
})

var Parent = Vue.extend({
    template: '...',
    components: {
         'my-component': Child
    }
})
```

Thiw new component () sarà disponibile solo all'interno dell'ambito (template) del componente Parent.

Registrazione in linea

Puoi estendere e registrare un componente in un solo passaggio:

```
Vue.component('custom-component', {
    template: '<div>A custom component!</div>'
})
```

Anche quando il componente è registrato localmente:

```
var Parent = Vue.extend({
    components: {
        'custom-component': {
            template: '<div>A custom component!</div>'
        }
    }
})
```

Registrazione dei dati nei componenti

Passare un oggetto alla proprietà dei `data` durante la registrazione di un componente farebbe sì che tutte le istanze del componente puntino agli stessi dati. Per risolvere questo, abbiamo bisogno di restituire i `data` da una funzione.

```
var CustomComponent = Vue.extend({
    data: function () {
        return { a: 1 }
    }
})
```

eventi

Uno dei modi in cui i componenti possono comunicare con i suoi antenati / discendenti è tramite eventi di comunicazione personalizzati. Tutte le istanze di Vue sono anche emettitori e implementano un'interfaccia evento personalizzata che facilita la comunicazione all'interno di un albero di componenti. Possiamo usare il seguente:

- `$on` : ascolta gli eventi emessi da questi componenti antenati o discendenti.
- `$broadcast` : emette un evento che si propaga verso il basso a tutti i discendenti.
- `$dispatch` : emette un evento che si innesca prima sul componente stesso e che si propaga verso l'alto a tutti gli antenati.
- `$emit` : attiva un evento su se stesso.

Ad esempio, vogliamo nascondere un componente pulsante specifico all'interno di un componente del modulo quando il modulo viene inviato. Sull'elemento genitore:

```
var FormComponent = Vue.extend({
  // ...
  components: {
    ButtonComponent
  },
  methods: {
    onSubmit () {
        this.$broadcast('submit-form')
    }
  }
})
```

Sull'elemento figlio:

```
var FormComponent = Vue.extend({
  // ...
  events: {
    'submit-form': function () {
        console.log('I should be hiding');
    }
  }
})
```

Alcune cose da tenere a mente:

- Ogni volta che un evento trova un componente che lo ascolta e viene attivato, interrompe la

propagazione a meno che la funzione callback in questo componente non restituisca `true` .

- `$dispatch()` si attiva sempre prima sul componente che lo ha emesso.
- Possiamo passare qualsiasi numero di argomenti al gestore di eventi. Facendo `this.$broadcast('submit-form', this.formData, this.formStatus)` ci permette di accedere a questi argomenti come `'submit-form': function (formData, formStatus) {}`

Leggi componenti online: https://riptutorial.com/it/vue-js/topic/1775/componenti

Capitolo 5: Componenti dinamici

Osservazioni

`<component>` è un elemento del componente riservato, non confonderlo con l'istanza dei componenti.

`v-bind` è una direttiva. Le direttive hanno come prefisso *v-* per indicare che sono attributi speciali forniti da Vue.

Examples

Esempio di componenti dinamici semplici

Passa dinamicamente tra più componenti usando l'elemento `<component>` e passa i dati a *v-bind: è l'* attributo:

Javascript:

```
new Vue({
  el: '#app',
  data: {
    currentPage: 'home'
  },
  components: {
    home: {
      template: "<p>Home</p>"
    },
    about: {
      template: "<p>About</p>"
    },
    contact: {
      template: "<p>Contact</p>"
    }
  }
})
```

HTML:

```
<div id="app">
   <component v-bind:is="currentPage">
       <!-- component changes when currentPage changes! -->
       <!-- output: Home -->
   </component>
</div>
```

Frammento:

Dimostrazione dal vivo

Pagine di navigazione con keep-alive

A volte vuoi tenere in memoria i componenti fuori-uscita, per far si che ciò accada, devi usare l'elemento `<keep-alive>` :

Javascript:

```
new Vue({
  el: '#app',
  data: {
    currentPage: 'home',
  },
  methods: {
     switchTo: function(page) {
            this.currentPage = page;
     }
  },
  components: {
    home: {
      template: `<div>
                 <h2>Home</h2>
                 <p>{{ homeData }}</p>
                 </div>`,
      data: function() {
         return {
            homeData: 'My about data'
         }
       }
     },
    about: {
      template: `<div>
                 <h2>About</h2>
                 <p>{{ aboutData }}</p>
                 </div>`,
      data: function() {
         return {
            aboutData: 'My about data'
         }
      }
    },
    contact: {
      template: `<div>
             <h2>Contact</h2>
             <form method="POST" @submit.prevent>
             <label>Your Name:</label>
             <input type="text" v-model="contactData.name" >
             <label>You message: </label>
             <textarea v-model="contactData.message"></textarea>
             <button type="submit">Send</button>
             </form>
             </div>`,
       data: function() {
          return {
            contactData: { name:'', message:'' }
         }
      }
```

```
    }
  }
})
```

HTML:

```
<div id="app">
  <div class="navigation">
    <ul>
      <li><a href="#home" @click="switchTo('home')">Home</a></li>
      <li><a href="#about" @click="switchTo('about')">About</a></li>
      <li><a href="#contact" @click="switchTo('contact')">Contact</a></li>
    </ul>
  </div>

  <div class="pages">
    <keep-alive>
      <component :is="currentPage"></component>
    </keep-alive>
  </div>
</div>
```

CSS:

```
.navigation {
  margin: 10px 0;
}

.navigation ul {
  margin: 0;
  padding: 0;
}

.navigation ul li {
  display: inline-block;
  margin-right: 20px;
}

input, label, button {
  display: block
}

input, textarea {
  margin-bottom: 10px;
}
```

Frammento:

Dimostrazione dal vivo

Leggi Componenti dinamici online: https://riptutorial.com/it/vue-js/topic/7702/componenti-dinamici

Capitolo 6: Componenti personalizzati con v-model

introduzione

Spesso dobbiamo creare alcuni componenti che eseguono alcune azioni / operazioni sui dati e ne richiediamo nel componente principale. La maggior parte delle volte `vuex` sarebbe una soluzione migliore, ma nei casi in cui il comportamento del componente figlio non ha nulla a che fare con lo stato dell'applicazione, ad esempio: un dispositivo di scorrimento intervallo, data / ora, lettore di file

Avere singoli negozi per ogni componente ogni volta che vengono utilizzati diventa complicato.

Osservazioni

Per avere il `v-model` su un componente devi soddisfare due condizioni.

1. Dovrebbe avere un sostegno chiamato "valore"
2. Dovrebbe emettere un evento di `input` con il valore previsto dai componenti principali.

```
<component v-model='something'></component>
```

è solo zucchero sintattico per

```
<component
    :value="something"
    @input="something = $event.target.value"
>
</component>
```

Examples

v-model su un contatore

Qui `counter` è un componente figlio a cui si accede tramite `demo` che è un componente principale che utilizza `v-model` .

```
// child component
Vue.component('counter', {
  template: `<div><button @click='add'>+1</button>
  <button @click='sub'>-1</button>
  <div>this is inside the child component: {{ result }}</div></div>`,
  data () {
    return {
      result: 0
```

```
    }
  },
  props: ['value'],
  methods: {
    emitResult () {
      this.$emit('input', this.result)
    },
    add () {
      this.result += 1
      this.emitResult()
    },
    sub () {
      this.result -= 1
      this.emitResult()
    }
  }
})
```

Questo componente figlio emetterà il `result` ogni volta che vengono chiamati i metodi `sub()` o `add()` .

```
// parent component
new Vue({
  el: '#demo',
  data () {
    return {
      resultFromChild: null
    }
  }
})

// parent template
<div id='demo'>
  <counter v-model='resultFromChild'></counter>
  This is in parent component {{ resultFromChild }}
</div>
```

Dato che `v-model` è presente sul componente figlio, è stata inviata una puntello con `value` nome allo stesso tempo, c'è un evento di input sul `counter` che a sua volta fornisce il valore dal componente figlio.

Leggi Componenti personalizzati con v-model online: https://riptutorial.com/it/vue-js/topic/9353/componenti-personalizzati-con-v-model

Capitolo 7: Direttive personalizzate

Sintassi

- `Vue.directive(id, definition);`
- `Vue.directive(id, update); //when you need only the update function.`

Parametri

Parametro	Dettagli
`id`	String - L'id direttiva che verrà utilizzato senza il prefisso `v-` . (Aggiungi il prefisso `v-` quando lo usi)
`definition`	Oggetto: un oggetto definizione può fornire diverse funzioni di hook (tutte facoltative): `bind` , `update` e `unbind`

Examples

Nozioni di base

Oltre al set predefinito di direttive fornite in core, Vue.js ti consente anche di registrare le direttive personalizzate. Le direttive personalizzate forniscono un meccanismo per mappare le modifiche dei dati al comportamento DOM arbitrario.

È possibile registrare una direttiva personalizzata globale con il `Vue.directive(id, definition)` , passando un id direttiva seguito da un oggetto definizione. È inoltre possibile registrare una direttiva personalizzata locale includendola nell'opzione delle `directives` un componente.

Funzioni di aggancio

- **bind** : chiamato una sola volta, quando la direttiva viene prima associata all'elemento.
- **aggiornamento** : chiamato per la prima volta immediatamente dopo il `bind` con il valore iniziale, quindi di nuovo ogni volta che cambia il valore di `bind` . Il nuovo valore e il valore precedente vengono forniti come argomento.
- **unbind** : chiamato solo una volta, quando la direttiva non è associata all'elemento.

```
Vue.directive('my-directive', {
     bind: function () {
       // do preparation work
       // e.g. add event listeners or expensive stuff
       // that needs to be run only once
     },
     update: function (newValue, oldValue) {
       // do something based on the updated value
       // this will also be called for the initial value
     },
```

```
    unbind: function () {
      // do clean up work
      // e.g. remove event listeners added in bind()
    }
})
```

Una volta registrato, puoi usarlo nei template Vue.js come questo (ricorda di aggiungere il prefisso `v-`):

```
<div v-my-directive="someValue"></div>
```

Quando hai solo bisogno della funzione di `update` , puoi passare una singola funzione invece dell'oggetto di definizione:

```
Vue.directive('my-directive', function (value) {
  // this function will be used as update()
})
```

Proprietà istanza direttiva

Tutte le funzioni hook verranno copiati nell'oggetto direttiva attuale, che è possibile accedere all'interno di queste funzioni come loro `this` contesto. L'oggetto direttiva espone alcune proprietà utili:

- **el** : l'elemento a cui è vincolata la direttiva.
- **vm** : il contesto ViewModel che possiede questa direttiva.
- **espressione** : l'espressione del bind, escludendo argomenti e filtri.
- **arg** : l'argomento, se presente.
- **nome** : il nome della direttiva, senza il prefisso.
- **modificatori** : un oggetto contenente modificatori, se presenti.
- **descrittore** : un oggetto che contiene il risultato di analisi dell'intera direttiva.
- **params:** un oggetto contenente attributi param. Spiegato di seguito.

  È necessario trattare tutte queste proprietà come di sola lettura e non modificarle mai. È possibile allegare anche proprietà personalizzate all'oggetto direttivo, ma attenzione a non sovrascrivere accidentalmente quelle interne esistenti.

Un esempio di una direttiva personalizzata che utilizza alcune di queste proprietà:

HTML

```
<div id="demo" v-demo:hello.a.b="msg"></div>
```

JavaScript

```
Vue.directive('demo', {
  bind: function () {
    console.log('demo bound!')
  },
  update: function (value) {
```

```
    this.el.innerHTML =
      'name - '       + this.name + '<br>' +
      'expression - ' + this.expression + '<br>' +
      'argument - '   + this.arg + '<br>' +
      'modifiers - '  + JSON.stringify(this.modifiers) + '<br>' +
      'value - '      + value
  }
})
var demo = new Vue({
  el: '#demo',
  data: {
    msg: 'hello!'
  }
})
```

Risultato

```
name - demo
expression - msg
argument - hello
modifiers - {"b":true,"a":true}
value - hello!
```

Oggetto letterale

Se la tua direttiva ha bisogno di più valori, puoi anche passare un valore letterale dell'oggetto JavaScript. Ricorda che le direttive possono assumere qualsiasi espressione JavaScript valida:

HTML

```
<div v-demo="{ color: 'white', text: 'hello!' }"></div>
```

JavaScript

```
Vue.directive('demo', function (value) {
  console.log(value.color) // "white"
  console.log(value.text) // "hello!"
})
```

Modificatore letterale

Quando una direttiva viene utilizzata con il modificatore letterale, il suo valore di attributo verrà interpretato come una stringa semplice e trasmesso direttamente nel metodo di `update` . Il metodo di `update` sarà anche chiamato una sola volta, perché una stringa semplice non può essere reattiva.

HTML

```
<div v-demo.literal="foo bar baz">
```

JavaScript

```
Vue.directive('demo', function (value) {
  console.log(value) // "foo bar baz"
})
```

Leggi Direttive personalizzate online: https://riptutorial.com/it/vue-js/topic/2368/direttive-personalizzate

Capitolo 8: Elenco di rendering

Examples

Uso di base

Un elenco può essere reso utilizzando la direttiva `v-for` . La sintassi richiede che venga specificato l'array di origine su cui eseguire l'iterazione e un alias che verrà utilizzato per fare riferimento a ciascun elemento nell'iterazione. Nell'esempio seguente vengono utilizzati `items` come matrice di origine e `item` come alias per ciascun elemento.

HTML

```
<div id="app">
  <h1>My List</h1>
  <table>
    <tr v-for="item in items">
      <td>{{item}}</td>
    </tr>
  </table>
</div>
```

copione

```
new Vue({
  el: '#app',
  data: {
    items: ['item 1', 'item 2', 'item 3']
  }
})
```

Puoi vedere una demo funzionante qui .

Mostra solo elementi HTML

In questo esempio verranno visualizzati cinque tag `<li>`

```
<ul id="render-sample">
  <li v-for="n in 5">
    Hello Loop
  </li>
</ul>
```

Elenco di conto alla rovescia di maiale

```
<ul>
  <li v-for="n in 10">{{11 - n}} pigs are tanning at the beach. One got fried, and
```

```
</ul>
```

https://jsfiddle.net/gurghet/3jeyka22/

Iterazione su un oggetto

`v-for` può essere usato per iterare su un oggetto chiavi (e valori):

HTML:

```
<div v-for="(value, key) in object">
  {{ key }} : {{ value }}
</div>
```

script:

```
new Vue({
  el: '#repeat-object',
  data: {
    object: {
      FirstName: 'John',
      LastName: 'Doe',
      Age: 30
    }
  }
})
```

Leggi Elenco di rendering online: https://riptutorial.com/it/vue-js/topic/1972/elenco-di-rendering

Capitolo 9: eventi

Examples

Sintassi degli eventi

Per inviare un evento: `vm.$emit('new-message');`

Per catturare un evento: `vm.$on('new-message');`

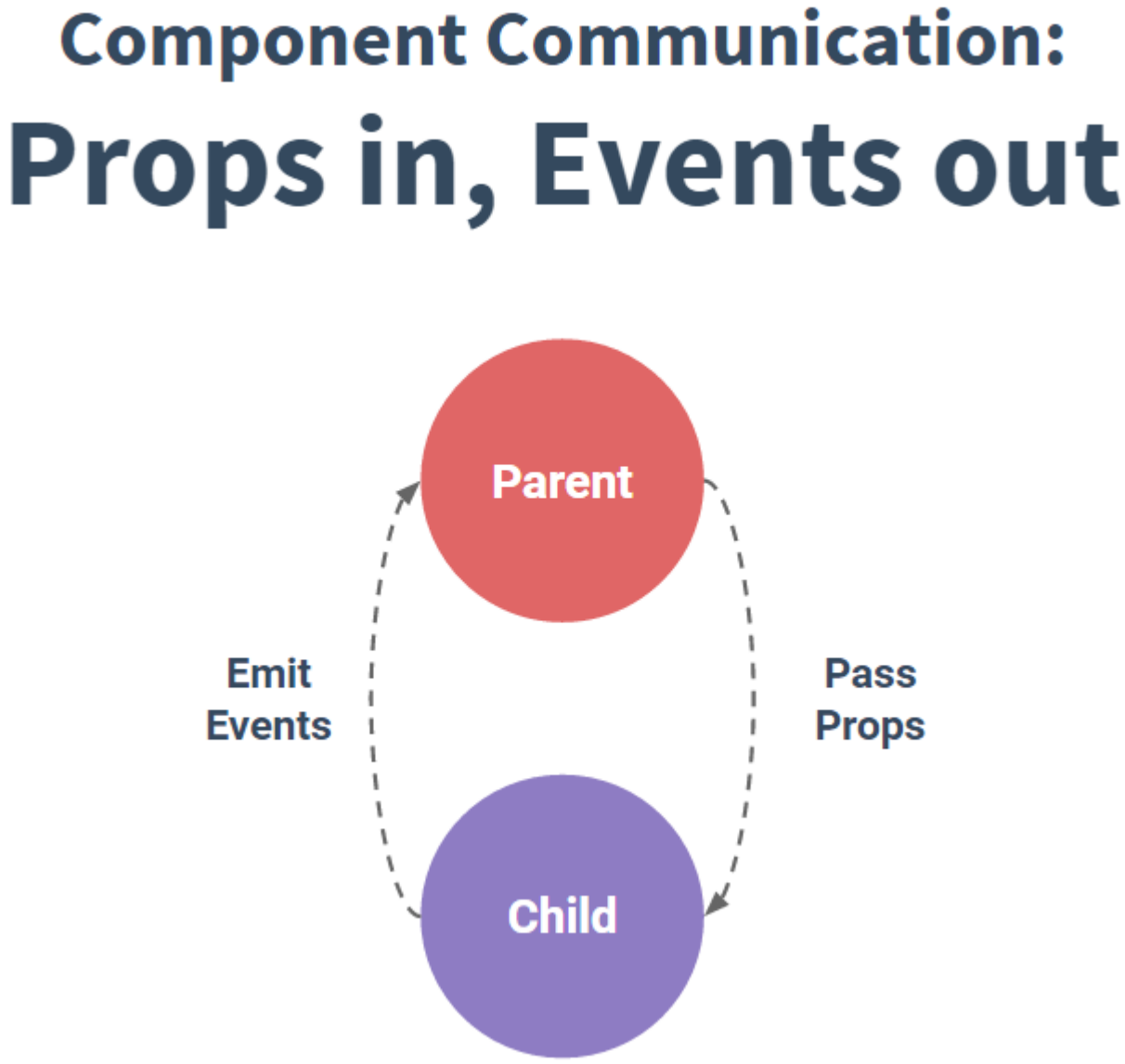
Per inviare un evento a tutti i componenti in **basso** : `vm.$broadcast('new-message');`

Per inviare un evento a tutti i componenti **up** : `vm.$dispatch('new-message');`

Nota: `$broadcast` e `$dispatch` sono deprecati in Vue2. (vedi le funzionalità di Vue2)

Quando dovrei usare gli eventi?

La seguente immagine illustra come dovrebbe funzionare la comunicazione dei componenti. L'immagine proviene dalle diapositive di Progressive Framework di Evan You (sviluppatore di VueJS).



Ecco un esempio di come funziona:

DEMO

HTML

```
<script type="x-template" id="message-box">
    <input type="text" v-model="msg" @keyup="$emit('new-message', msg)" />
</script>

<message-box :msg="message" @new-message="updateMessage"></message-box>
<div>You typed: {{message}}</div>
```

JS

```
var messageBox = {
  template: '#message-box',
  props: ['msg']
};

new Vue({
  el: 'body',
```

```
  data: {
    message: ''
  },
  methods: {
    updateMessage: function(msg) {
      this.message = msg;
    }
  },
  components: {
    'message-box': messageBox
  }
});
```

L'esempio sopra può essere migliorato!

L'esempio sopra mostra come funziona la comunicazione del componente. Ma nel caso di un componente di input personalizzato, per sincronizzare la variabile padre con il valore digitato, dovremmo usare `v-model` .

DEMO Vue1

DEMO Vue2

In Vue1, dovresti usare `.sync` sul puntello inviato al componente `<message-box>` . Questo dice a VueJS di sincronizzare il valore nel componente figlio con quello del genitore.

> **Ricorda:** ogni istanza del componente ha il proprio ambito isolato.

HTML Vue1

```
<script type="x-template" id="message-box">
  <input v-model="value" />
</script>

<div id="app">
  <message-box :value.sync="message"></message-box>
  <div>You typed: {{message}}</div>
</div>
```

In Vue2, c'è un evento speciale `'input'` che puoi `$emit` . L'utilizzo di questo evento ti consente di mettere un `v-model` direttamente sul componente `<message-box>` . L'esempio avrà il seguente aspetto:

HTML Vue2

```
<script type="x-template" id="message-box">
  <input :value="value" @input="$emit('input', $event.target.value)" />
</script>

<div id="app">
  <message-box v-model="message"></message-box>
  <div>You typed: {{message}}</div>
</div>
```

JS Vue 1 & 2

```
var messageBox = {
  template: '#message-box',
  props: ['value']
};

new Vue({
  el: '#app',
  data: {
    message: ''
  },
  components: {
    'message-box': messageBox
  }
});
```

Nota quanto più veloce è l'aggiornamento dell'aggiornamento.

Come gestire la deprecazione di $ spedizione e $ broadcast? (modello di evento bus)

Potresti aver capito che `$emit` come ambito il componente che sta emettendo l'evento. Questo è un problema quando si desidera comunicare tra componenti distanti l'uno dall'altro nell'albero dei componenti.

Nota: in Vue1 si usa `$dispatch` o `$broadcast` , ma non in Vue2. Il motivo è che non scala bene. C'è un modello di `bus` popolare per gestirlo:

DEMO

HTML

```
<script type="x-template" id="sender">
  <button @click="bus.$emit('new-event')">Click me to send an event !</button>
</script>

<script type="x-template" id="receiver">
  <div>I received {{numberOfEvents}} event{{numberOfEvents == 1 ? '' : 's'}}</div>
</script>

<sender></sender>
<receiver></receiver>
```

JS

```
var bus = new Vue();

var senderComponent = {
  template: '#sender',
  data() {
    return {
      bus: bus
    }
```

```
  }
};

var receiverComponent = {
  template: '#receiver',
  data() {
    return {
      numberOfEvents: 0
    }
  },
  ready() {
    var self = this;

    bus.$on('new-event', function() {
      ++self.numberOfEvents;
    });
  }
};

new Vue({
  el: 'body',
  components: {
    'sender': senderComponent,
    'receiver': receiverComponent
  }
});
```

Devi solo capire che ogni istanza di `Vue()` può `$emit` e prendere (`$on`) un evento. Dichiariamo solo un `bus` chiamata di istanze `Vue` globale e quindi qualsiasi componente con questa variabile può emettere e catturare eventi da esso. Assicurati solo che il componente abbia accesso alla variabile del `bus` .

Leggi eventi online: https://riptutorial.com/it/vue-js/topic/5941/eventi

Capitolo 10: Filtri personalizzati

Sintassi

- `Vue.filter(name, function(value){});` //Di base
- `Vue.filter(name, function(value, begin, end){});` // Base con valori di avvolgimento
- `Vue.filter(name, function(value, input){});` //Dinamico
- `Vue.filter(name, { read: function(value){}, write: function(value){} });` // due vie

Parametri

Parametro	Dettagli
nome	String - nome callable desiderato del filtro
valore	[Callback] Qualsiasi valore dei dati che passano nel filtro
inizio	[Callback] Qualsiasi - valore in arrivo prima dei dati passati
fine	[Callback] Qualsiasi valore che verrà dopo i dati passati
ingresso	[Callback] Qualsiasi - input utente associato all'istanza Vue per risultati dinamici

Examples

Filtri a due vie

Con un `two-way filter` , siamo in grado di assegnare un'operazione di `read` *e* `write` per un singolo `filter` che modifica il valore degli *stessi* dati tra la `view` e il `model` .

```
//JS
Vue.filter('uppercase', {
    //read : model -> view
    read: function(value) {
        return value.toUpperCase();
    },

    //write : view -> model
    write: function(value) {
        return value.toLowerCase();
    }
});

/*
 * Base value of data: 'example string'
 *
 * In the view : 'EXAMPLE STRING'
 * In the model : 'example string'
```

```
 */
```

Di base

I filtri personalizzati in `Vue.js` possono essere creati facilmente in una singola chiamata a `Vue.filter` .

```
//JS
Vue.filter('reverse', function(value) {
    return value.split('').reverse().join('');
});

//HTML
<span>{{ msg | reverse }}</span> //'This is fun!' => '!nuf si sihT'
```

È buona norma memorizzare tutti i filtri personalizzati in file separati, ad es. Sotto `./filters` in quanto è facile riutilizzare il codice nella prossima applicazione. Se vai in questo modo devi **sostituire la parte JS** :

```
//JS
Vue.filter('reverse', require('./filters/reverse'));
```

È inoltre possibile definire i propri `begin` e `end` involucri pure.

```
//JS
Vue.filter('wrap', function(value, begin, end) {
    return begin + value + end;
});

//HTML
<span>{{ msg | wrap 'The' 'fox' }}</span> //'quick brown' => 'The quick brown fox'
```

Leggi Filtri personalizzati online: https://riptutorial.com/it/vue-js/topic/1878/filtri-personalizzati

Capitolo 11: L'array modifica le avvertenze sul rilevamento

introduzione

Quando si tenta di impostare un valore di un elemento in un determinato indice di un array inizializzato nell'opzione dati, Vue non può rilevare la modifica e non attiva un aggiornamento allo stato. Per superare questa avvertenza dovresti utilizzare Vue di vue. $ Set o usare il metodo Array.prototype.splice

Examples

Usando Vue. $ Set

Nel metodo o in qualsiasi hook del ciclo di vita che modifica l'elemento dell'array in un indice particolare

```
new Vue({
    el: '#app',
    data:{
        myArr : ['apple', 'orange', 'banana', 'grapes']
    },
    methods:{
        changeArrayItem: function(){
            //this will not work
            //myArr[2] = 'strawberry';

            //Vue.$set(array, index, newValue)
            this.$set(this.myArr, 2, 'strawberry');
        }
    }
})
```

Ecco il link al violino

Utilizzando Array.prototype.splice

È possibile eseguire la stessa modifica anziché utilizzare `Vue.$set` utilizzando la `splice()` del prototipo di Array `splice()`

```
new Vue({
    el: '#app',
    data:{
        myArr : ['apple', 'orange', 'banana', 'grapes']
    },
    methods:{
        changeArrayItem: function(){
            //this will not work
            //myArr[2] = 'strawberry';
```

```
            //Array.splice(index, 1, newValue)
            this.myArr.splice(2, 1, 'strawberry');
        }
    }
})
```

Per matrice nidificata

Se hai una matrice annidata, puoi fare quanto segue

```
new Vue({
    el: '#app',
    data:{
        myArr : [
            ['apple', 'banana'],
            ['grapes', 'orange']
        ]
    },
    methods:{
        changeArrayItem: function(){
            this.$set(this.myArr[1], 1, 'strawberry');
        }
    }
})
```

Ecco il link al jsfiddle

Matrice di oggetti contenenti matrici

```
new Vue({
    el: '#app',
    data:{
        myArr : [
            {
                name: 'object-1',
                nestedArr: ['apple', 'banana']
            },
            {
                name: 'object-2',
                nestedArr: ['grapes', 'orange']
            }
        ]
    },
    methods:{
        changeArrayItem: function(){
            this.$set(this.myArr[1].nestedArr, 1, 'strawberry');
        }
    }
})
```

Ecco il link al violino

Leggi L'array modifica le avvertenze sul rilevamento online: https://riptutorial.com/it/vue-js/topic/10679/l-array-modifica-le-avvertenze-sul-rilevamento

Capitolo 12: Lifecycle Hooks

Examples

Ganci per Vue 1.x

- **`init`**

  Chiamato in modo sincrono dopo l'inizializzazione dell'istanza e prima di qualsiasi osservazione iniziale dei dati.

- **`created`**

  Chiamato in modo sincrono dopo la creazione dell'istanza. Ciò si verifica prima di `$el` setup, ma dopo `data observation` , sono state configurate `computed properties` , `watch/event callbacks` e `methods` .

- **`beforeCompile`**

  Immediatamente prima della compilazione dell'istanza Vue.

- **`compiled`**

  Subito dopo la compilazione è stata completata. Tutte le `directives` sono collegate ma ancora prima che `$el` sia disponibile.

- **`ready`**

  Si verifica dopo la compilazione *e* `$el` sono completi e l'istanza viene iniettata nel DOM per la prima volta.

- **`attached`**

  Si verifica quando `$el` è collegato al DOM da una `directive` o un'istanza chiama `$appendTo()` .

- **`detached`**

  Chiamato quando `$el` viene rimosso / staccato dal DOM o dal metodo di istanza.

- **`beforeDestroy`**

  Immediatamente prima che l'istanza di Vue venga distrutta, ma è ancora completamente funzionante.

- **`destroyed`**

  Chiamato dopo che un'istanza è stata distrutta. Tutti i `bindings` e le `directives` sono già stati

non vincolati e anche le istanze secondarie sono state distrutte.

Uso in un'istanza

Poiché *tutti gli* hook del ciclo di vita in `Vue.js` sono solo `functions` , è possibile `Vue.js` *uno* direttamente nell'istanza dichiaraction.

```
//JS
new Vue({

    el: '#example',

    data: {
        ...
    },

    methods: {
        ...
    },

    //LIFECYCLE HOOK HANDLING
    created: function() {
        ...
    },

    ready: function() {
        ...
    }

});
```

Common Trappole: Accesso al DOM dal hook `ready ()`

Un usecase comune per l'hook `ready()` è quello di accedere al DOM, ad esempio per avviare un plugin Javascript, ottenere le dimensioni di un elemento, ecc.

Il problema

A causa del meccanismo di aggiornamento DOM asincrono di Vue, non è garantito che il DOM sia stato completamente aggiornato quando viene chiamato l'hook `ready()` . Di solito ciò genera un errore perché l'elemento non è definito.

La soluzione

Per questa situazione, il metodo di istanza `$nextTick()` può essere d'aiuto. Questo metodo rimanda l'esecuzione della funzione di callback fornita fino a dopo il segno di spunta successivo, il che significa che viene attivato quando tutti gli aggiornamenti DOM sono garantiti per essere completato.

Esempio:

```
module.exports {
  ready: function () {
```

```
    $('.cool-input').initiateCoolPlugin() //fails, because element is not in DOM yet.

    this.$nextTick(function() {
      $('.cool-input').initiateCoolPlugin() // this will work because it will be executed
after the DOM update.
    })
  }
}
```

Leggi Lifecycle Hooks online: https://riptutorial.com/it/vue-js/topic/1852/lifecycle-hooks

Capitolo 13: mixins

Examples

Mixin globale

Puoi anche applicare un mixin a livello globale. Fai attenzione! Una volta applicato un mixin a livello globale, questo influenzerà **ogni** istanza di Vue creata in seguito. Se utilizzato correttamente, può essere utilizzato per iniettare la logica di elaborazione per le opzioni personalizzate:

```
// inject a handler for `myOption` custom option
Vue.mixin({
  created: function () {
    var myOption = this.$options.myOption
    if (myOption) {
      console.log(myOption)
    }
  }
})

new Vue({
  myOption: 'hello!'
})
// -> "hello!"
```

> Utilizza i mixaggi globali con parsimonia e attenzione, poiché influisce su ogni singola istanza di Vue creata, compresi i componenti di terze parti. Nella maggior parte dei casi, dovresti usarlo solo per la gestione delle opzioni personalizzate come dimostrato nell'esempio sopra.

Strategie di fusione personalizzate

Quando le opzioni personalizzate vengono unite, utilizzano la strategia predefinita, che sovrascrive semplicemente il valore esistente. Se si desidera unire l'opzione personalizzata utilizzando la logica personalizzata, è necessario associare una funzione a `Vue.config.optionMergeStrategies` :

```
Vue.config.optionMergeStrategies.myOption = function (toVal, fromVal) {
  // return mergedVal
}
```

Per la maggior parte delle opzioni basate su oggetti, puoi semplicemente utilizzare la stessa strategia utilizzata dai `methods` :

```
var strategies = Vue.config.optionMergeStrategies
strategies.myOption = strategies.methods
```

Nozioni di base

I mix sono un modo flessibile per distribuire funzionalità riutilizzabili per i componenti Vue. Un oggetto mixin può contenere qualsiasi opzione di componente. Quando un componente usa un mixin, tutte le opzioni nel mixin saranno "mixate" nelle opzioni del componente stesso.

```
// define a mixin object
var myMixin = {
  created: function () {
    this.hello()
  },
  methods: {
    hello: function () {
      console.log('hello from mixin!')
    }
  }
}

// define a component that uses this mixin
var Component = Vue.extend({
  mixins: [myMixin]
})

var component = new Component() // -> "hello from mixin!"
```

Opzione di unione

Quando un mixin e il componente stesso contengono opzioni sovrapposte, saranno "uniti" usando strategie appropriate. Ad esempio, le funzioni di hook con lo stesso nome vengono unite in una matrice in modo che vengano chiamate tutte. Inoltre, i hook di mixin verranno chiamati **prima** degli hook del componente:

```
var mixin = {
  created: function () {
    console.log('mixin hook called')
  }
}

new Vue({
  mixins: [mixin],
  created: function () {
    console.log('component hook called')
  }
})

// -> "mixin hook called"
// -> "component hook called"
```

Le opzioni che prevedono valori di oggetto, ad esempio `methods` , `components` e `directives` , verranno unite nello stesso oggetto. Le opzioni del componente avranno la priorità quando ci sono chiavi in conflitto in questi oggetti:

```
var mixin = {
  methods: {
```

```
    foo: function () {
      console.log('foo')
    },
    conflicting: function () {
      console.log('from mixin')
    }
  }
}

var vm = new Vue({
  mixins: [mixin],
  methods: {
    bar: function () {
      console.log('bar')
    },
    conflicting: function () {
      console.log('from self')
    }
  }
})

vm.foo() // -> "foo"
vm.bar() // -> "bar"
vm.conflicting() // -> "from self"
```

Si noti che le stesse strategie di unione vengono utilizzate in `Vue.extend()` .

Leggi mixins online: https://riptutorial.com/it/vue-js/topic/2562/mixins

Capitolo 14: Modello "webpack" di Polyfill

Parametri

File o pacchetti	Comando o configurazione da modificare
babel-polyfill	`npm i -save babel-polyfill`
karma.conf.js	`files: ['../../node_modules/babel-polyfill/dist/polyfill.js','./index.js'],`
webpack.base.conf.js	`app: ['babel-polyfill', './src/main.js']`

Osservazioni

Le configurazioni descritte sopra, l'esempio che utilizza una funzione non standardizzata funzionerà su "internet explorer" e il `npm test` passerà.

Examples

Utilizzo delle funzioni su polyfill (es: trovare)

```
<template>
  <div class="hello">
    <p>{{ filtered() }}</p>
  </div>
</template>

<script>
export default {
  name: 'hello',
  data () {
    return {
      list: ['toto', 'titi', 'tata', 'tete']
    }
  },
  methods: {
    filtered () {
      return this.list.find((el) => el === 'tata')
    }
  }
}
</script>
```

Leggi Modello "webpack" di Polyfill online: https://riptutorial.com/it/vue-js/topic/9174/modello--webpack--di-polyfill

Capitolo 15: modificatori

introduzione

Ci sono alcune operazioni usate frequentemente come `event.preventDefault()` o `event.stopPropagation()` all'interno dei gestori di eventi. Anche se possiamo farlo facilmente all'interno dei metodi, sarebbe meglio se i metodi potessero essere puramente basati sulla logica dei dati piuttosto che dover gestire i dettagli dell'evento DOM.

Examples

Modificatori di eventi

Vue fornisce i modificatori di eventi per `v-on` chiamando i suffissi di direttive denotati da un punto.

- `.stop`
- `.prevent`
- `.capture`
- `.self`
- `.once`

Per esempio:

```
<!-- the click event's propagation will be stopped -->
<a v-on:click.stop="doThis"></a>

<!-- the submit event will no longer reload the page -->
<form v-on:submit.prevent="onSubmit"></form>

<!-- use capture mode when adding the event listener -->
<div v-on:click.capture="doThis">...</div>

<!-- only trigger handler if event.target is the element itself -->
<!-- i.e. not from a child element -->
<div v-on:click.self="doThat">...</div>
```

Modificatori chiave

Quando si ascoltano eventi di tastiera, spesso è necessario verificare i codici chiave comuni. Ricordare tutti i keycode è una seccatura, quindi Vue fornisce alias per le chiavi più comunemente usate:

- `.enter`
- `.tab`
- `.delete` (cattura entrambi i tasti "Cancella" e "Backspace")
- `.esc`
- `.space`
- `.up`
- `.down`

- `.left`
- `.right`

Per esempio:

```
<input v-on:keyup.enter="submit">
```

Modificatori di input

- `.trim`

Se si desidera che l'input dell'utente venga ritagliato automaticamente, è possibile aggiungere il modificatore di `trim` agli input gestiti del `v-model` :

```
<input v-model.trim="msg">
```

- `.number`

Se vuoi che l'input dell'utente venga automaticamente tipizzato come un numero, puoi fare come segue:

```
<input v-model.number="age" type="number">
```

- `.lazy`

Generalmente, `v-model` sincronizza l'input con i dati dopo ogni evento di input, ma è possibile aggiungere il modificatore `lazy` per sincronizzarsi dopo gli eventi di modifica:

```
<input v-model.lazy="msg" >
```

Leggi modificatori online: https://riptutorial.com/it/vue-js/topic/8612/modificatori

Capitolo 16: osservatori

Examples

Come funziona

Puoi guardare le proprietà dei dati di qualsiasi istanza di Vue. Quando si guarda una proprietà, si attiva un metodo di modifica:

```
export default {
    data () {
        return {
            watched: 'Hello World'
        }
    },
    watch: {
        'watched' () {
            console.log('The watched property has changed')
        }
    }
}
```

Puoi recuperare il vecchio valore e quello nuovo:

```
export default {
    data () {
        return {
            watched: 'Hello World'
        }
    },
    watch: {
        'watched' (value, oldValue) {
            console.log(oldValue) // Hello World
            console.log(value) // ByeBye World
        }
    },
    mounted () {
        this.watched = 'ByeBye World'
    }
}
```

Se hai bisogno di vedere le proprietà annidate su un oggetto, dovrai utilizzare la proprietà `deep` :

```
export default {
    data () {
        return {
            someObject: {
                message: 'Hello World'
            }
        }
    },
    watch: {
        'someObject': {
```

```
            deep: true,
            handler (value, oldValue) {
                console.log('Something changed in someObject')
            }
        }
    }
}
```

Quando vengono aggiornati i dati?

Se è necessario attivare l'osservatore prima di apportare alcune nuove modifiche a un oggetto, è necessario utilizzare il metodo `nextTick()` :

```
export default {
    data() {
        return {
            foo: 'bar',
            message: 'from data'
        }
    },
    methods: {
        action () {
            this.foo = 'changed'
            // If you juste this.message = 'from method' here, the watcher is executed after.
            this.$nextTick(() => {
                this.message = 'from method'
            })
        }
    },
    watch: {
        foo () {
            this.message = 'from watcher'
        }
    }
}
```

Leggi osservatori online: https://riptutorial.com/it/vue-js/topic/7988/osservatori

Capitolo 17: plugin

introduzione

I plug-in Vue aggiungono funzionalità globali come, metodi globali, direttive, transizioni, filtri, metodi di istanza, oggetti e iniettano alcune opzioni di componenti usando mixin

Sintassi

- MyPlugin.install = function (Vue, options) {}

Parametri

Nome	Descrizione
Vue	Costruttore Vue, iniettato da Vue
opzioni	Opzioni aggiuntive se necessario

Osservazioni

Nella maggior parte dei casi dovrai dire esplicitamente a Vue di usare un plugin

```
// calls `MyPlugin.install(Vue)`
Vue.use(MyPlugin)
```

Per passare opzioni

```
Vue.use(MyPlugin, { someOption: true })
```

Examples

Logger semplice

```
//myLogger.js
export default {

  install(Vue, options) {
     function log(type, title, text) {
       console.log(`[${type}] ${title} - ${text}`);
     }

     Vue.prototype.$log = {
       error(title, text) { log('danger', title, text) },
       success(title, text) { log('success', title, text) },
```

```
        log
      }
  }
}
```

Prima che l'istanza di Vue principale comunichi di registrare il tuo plugin

```
//main.js
import Logger from './path/to/myLogger';

Vue.use(Logger);

var vm = new Vue({
  el: '#app',
  template: '<App/>',
  components: { App }
})
```

Ora puoi chiamare `this.$log` a qualsiasi componente figlio

```
//myComponent.vue
export default {
  data() {
    return {};
  },
  methods: {
    Save() {
      this.$log.success('Transaction saved!');
    }
  }
}
```

Leggi plugin online: https://riptutorial.com/it/vue-js/topic/8726/plugin

Capitolo 18: Proprietà calcolate

Osservazioni

Dati vs Proprietà calcolate

La principale differenza del caso d'uso per i `data` e le proprietà `computed` di un'istanza `Vue` dipende dallo stato potenziale o dalla probabilità di modifica dei dati. Al momento di decidere quale categoria dovrebbe essere un determinato oggetto, queste domande potrebbero aiutare:

- È un valore costante? (**dati**)
- Ha la possibilità di cambiare? (**calcolato** o **dati**)
- Il suo valore dipende dal valore di altri dati? (**calcolato**)
- Ha bisogno di dati o calcoli aggiuntivi per essere completato prima di essere utilizzato? (**calcolato**)
- Il valore cambierà solo in determinate circostanze? (**dati**)

Examples

Esempio di base

Modello

```
<div id="example">
  a={{ a }}, b={{ b }}
</div>
```

JavaScript

```
var vm = new Vue({
  el: '#example',
  data: {
    a: 1
  },
  computed: {
    // a computed getter
    b: function () {
      // `this` points to the vm instance
      return this.a + 1
    }
  }
})
```

Risultato

```
a=1, b=2
```

Qui abbiamo dichiarato una proprietà calcolata `b` . La funzione che abbiamo fornito verrà utilizzata come funzione getter per la proprietà `vm.b` :

```
console.log(vm.b) // -> 2
vm.a = 2
console.log(vm.b) // -> 3
```

Il valore di `vm.b` dipende sempre dal valore di `vm.a`

È possibile associare i dati alle proprietà calcolate nei modelli proprio come una proprietà normale. Vue è consapevole del fatto che `vm.b` dipende da `vm.a` , quindi aggiornerà tutti i binding che dipendono da `vm.b` quando `vm.a` cambia.

Proprietà calcolate vs orologio

modello

```
<div id="demo">{{fullName}}</div>
```

Guarda l'esempio

```
var vm = new Vue({
  el: '#demo',
  data: {
    firstName: 'Foo',
    lastName: 'Bar',
    fullName: 'Foo Bar'
  }
})

vm.$watch('firstName', function (val) {
  this.fullName = val + ' ' + this.lastName
})

vm.$watch('lastName', function (val) {
  this.fullName = this.firstName + ' ' + val
})
```

Esempio calcolato

```
var vm = new Vue({
  el: '#demo',
  data: {
    firstName: 'Foo',
    lastName: 'Bar'
  },
  computed: {
    fullName: function () {
      return this.firstName + ' ' + this.lastName
    }
  }
})
```

Setter contornati

Le proprietà calcolate verranno automaticamente ricalcolate ogni volta che cambiano i dati da cui dipende il calcolo. Tuttavia, se è necessario modificare manualmente una proprietà calcolata, Vue ti consente di creare un metodo setter per farlo:

Modello (dall'esempio di base sopra):

```
<div id="example">
  a={{ a }}, b={{ b }}
</div>
```

Javascript:

```
var vm = new Vue({
  el: '#example',
  data: {
    a: 1
  },
  computed: {
    b: {
      // getter
      get: function () {
        return this.a + 1
      },
      // setter
      set: function (newValue) {
        this.a = newValue - 1
      }
    }
  }
```

Ora puoi invocare il getter o il setter:

```
console.log(vm.b)       // -> 2
vm.b = 4                // (setter)
console.log(vm.b)       // -> 4
console.log(vm.a)       // -> 3
```

`vm.b = 4` invocherà il setter e imposta this.a su 3; per estensione, vm.b valuterà a 4.

Usare setter calcolati per v-model

Potrebbe essere necessario un `v-model` su una proprietà calcolata. Normalmente, v-model non aggiornerà il valore della proprietà calcolata.

Il template:

```
<div id="demo">
  <div class='inline-block card'>
    <div :class='{onlineMarker: true, online: status, offline: !status}'></div>
    <p class='user-state'>User is {{ (status) ? 'online' : 'offline' }}</p>
```

```
  </div>

  <div class='margin-5'>
    <input type='checkbox' v-model='status'>Toggle status (This will show you as offline to
others)
  </div>
</div>
```

Messa in piega:

```
#demo {
  font-family: Helvetica;
  font-size: 12px;
}
.inline-block > * {
  display: inline-block;
}
.card {
  background: #ddd;
  padding:2px 10px;
  border-radius: 3px;
}
.onlineMarker {
  width: 10px;
  height: 10px;
  border-radius: 50%;
  transition: all 0.5s ease-out;
}

.online {
  background-color: #3C3;
}

.offline {
  background-color: #aaa;
}

.user-state {
  text-transform: uppercase;
  letter-spacing: 1px;
}

.margin-5 {
  margin: 5px;
}
```

Il componente:

```
var demo = new Vue({
    el: '#demo',
    data: {
        statusProxy: null
    },
    computed: {
        status: {
            get () {
                return (this.statusProxy === null) ? true : this.statusProxy
            }
        }
```

```
    }
})
```

Fiddle Qui vedresti , cliccando sul pulsante radio non ha alcun senso, il tuo stato è ancora online.

```
var demo = new Vue({
    el: '#demo',
    data: {
        statusProxy: null
    },
    computed: {
        status: {
            get () {
                return (this.statusProxy === null) ? true : this.statusProxy
            },
            set (val) {
                this.statusProxy = val
            }
        }
    }
})
```

violino E ora si può vedere la levetta avviene come la casella di controllo è selezionata / deselezionata.

Leggi Proprietà calcolate online: https://riptutorial.com/it/vue-js/topic/2371/proprieta-calcolate

Capitolo 19: puntelli

Osservazioni

camelCase <=> kebab-case

Quando definisci i nomi dei tuoi `props` di `props` , ricorda sempre che i nomi degli attributi HTML non fanno distinzione tra maiuscole e minuscole. Ciò significa che se si definisce un `prop` in caso di cammello nella definizione del componente ...

```
Vue.component('child', {
    props: ['myProp'],
    ...
});
```

... devi chiamarlo nel tuo componente HTML come mio prop.

Examples

Trasmissione dei dati da genitore a figlio con oggetti di scena

In Vue.js, ogni istanza del componente ha ***il proprio ambito isolato*** , il che significa che se un componente padre ha un componente figlio - il componente figlio ha il proprio ambito isolato e il componente padre ha il proprio ambito isolato.

Per qualsiasi app di dimensioni medio-grandi, le seguenti convenzioni sulle best practice evitano molti mal di testa durante la fase di sviluppo e successivamente la manutenzione. Una delle cose da seguire è quella di ***evitare di fare riferimento / mutare i dati principali direttamente dal componente figlio*** . Allora, come facciamo a fare riferimento ai dati principali all'interno di un componente figlio?

Qualsiasi dato genitore è richiesto in un componente figlio dovrebbe essere passato al bambino come `props` di `props` dal genitore.

Caso d'uso : supponiamo di avere un database utente con due tabelle `users` e `addresses` con i seguenti campi:
Tabella `users`

nome	Telefono	e-mail
John Mclane	(1) 234 5678 9012	john@dirhard.com
James Bond	(44) 777 0007 0077	bond@mi6.com

tabella degli `addresses`

bloccare	strada	città
Nakatomi Towers	Broadway	New York
Mi6 House	Buckingham Road	Londra

e vogliamo avere tre componenti per visualizzare le informazioni utente corrispondenti ovunque nella nostra app

user-Component.js

```
export default{
    template:`<div class="user-component">
                <label for="name" class="form-control">Name: </label>
                <input class="form-control input-sm" name="name" v-model="name">
                <contact-details :phone="phone" :email="email"></contact-details>
              </div>`,
    data(){
        return{
            name:'',
            phone:'',
            email:''
        }
    },
}
```

fornitura-details.js

```
import Address from './address';
export default{
    template:`<div class="contact-details-component>
                <h4>Contact Details:</h4>
                <label for="phone" class="form-control">Phone: </label>
                <input class="form-control input-sm" name="phone" v-model="phone">
                <label for="email" class="form-control">Email: </label>
                <input class="form-control input-sm" name="email" v-model="email">

                <h4>Address:</h4>
                <address :address-type="addressType"></address>
                //see camelCase vs kebab-case explanation below
            </div>`,
    props:['phone', 'email'],
    data:(){
        return:{
            addressType:'Office'
        }
    },
    components:{Address}
}
```

address.js

```
export default{
    template:`<div class="address-component">
                <h6>{{addressType}}</h6>
                <label for="block" class="form-control">Block: </label>
```

```
                <input class="form-control input-sm" name="block" v-model="block">
                <label for="street" class="form-control">Street: </label>
                <input class="form-control input-sm" name="street" v-model="street">
                <label for="city" class="form-control">City: </label>
                <input class="form-control input-sm" name="city" v-model="city">
            </div>`,
    props:{
        addressType:{
            required:true,
            type:String,
            default:'Office'
        },
    data(){
        return{
            block:'',
            street:'',
            city:''
        }
    }
}
```

main.js

```
import Vue from 'vue';

Vue.component('user-component', require'./user-component');
Vue.component('contact-details', require'./contact-details');

new Vue({
    el:'body'
});
```

index.html

```
...
<body>
    <user-component></user-component>
        ...
</body>
```

Stiamo visualizzando i dati relativi a `phone` ed e `email` , che sono proprietà del `user-component` nei `contact-details` che non hanno dati telefonici o di posta elettronica.

Trasmissione dei dati come oggetti di scena

Quindi all'interno di `user-component.js` nella proprietà ***template*** , dove includiamo il componente `<contact-details>` , passiamo il ***telefono*** e i dati ***dell'email*** da `<user-component>` (componente padre) a `<contact-details>` (componente bambino) legandosi in modo dinamico agli **oggetti di scena** - `:phone="phone"` e `:email="email` , che è uguale a `v-bind:phone="phone"` e `v-bind:email="email"`

Puntelli - Rilegatura dinamica

Dal momento che leghiamo in modo dinamico gli oggetti di scena, qualsiasi cambiamento nel ***telefono*** o nella ***posta elettronica*** all'interno del componente principale, ovvero `<user-component>`

si rifletterà immediatamente nel componente figlio, ovvero `<contact-details>` .

Puntelli - come letterali

Tuttavia, se avessimo passato i valori di ***telefono*** ed e ***-mail*** come valori letterali stringa come `phone="(44) 777 0007 0077" email="bond@mi6.com"` allora non rifletterebbero eventuali modifiche ai dati che si verificano nel genitore componente.

Associazione unidirezionale

Per impostazione predefinita, la direzione delle modifiche è dall'alto verso il basso, ovvero qualsiasi modifica agli oggetti puntati dinamicamente nel componente padre si propagherà al componente figlio ma qualsiasi modifica ai valori di prop in un componente figlio non si propagherà al padre.

Ad es .: se dall'interno di `<contact-details>` cambiamo l'e-mail da `bond@mi6.com` a `jamesbond@mi6.com` , i dati parent, cioè la proprietà dei dati del telefono in `<user-component>` conterranno ancora un valore di `bond@mi6.com` .

Tuttavia, se cambiamo il valore dell'email da `bond@mi6.com` a `jamesbond@mi6.co` nel componente padre (`<user-component>` componente `<user-component>` nel nostro caso d'uso), allora il valore dell'email nel componente figlio (`<contact-details>` nel nostro caso d'uso) cambierà automaticamente in `jamesbond@mi6.com` - il cambiamento in genitore viene istantaneamente propagato al bambino.

Associazione a due vie

Se vogliamo un legame a due vie, dobbiamo specificare esplicitamente l'associazione bidirezionale come `:email.sync="email"` invece di `:email="email"` . Ora se cambiamo il valore dell'elica nel componente figlio, la modifica si rifletterà anche nel componente principale.

In un'app di medie e grandi dimensioni, cambiare lo stato genitore dallo stato figlio sarà molto difficile da rilevare e tenere traccia soprattutto durante il debug - **Sii prudente** .

Non ci sarà alcuna opzione .sync disponibile in Vue.js 2.0. **Il binding a due vie per gli oggetti di scena è stato deprecato in Vue.js 2.0** .

Legatura unica

E 'anche possibile definire esplicitamente **una volta** vincolanti come `:email.once="email` , è più o meno simile al passaggio di un letterale, perché eventuali successive variazioni del valore della proprietà genitore non si propagheranno al bambino.

AVVERTIMENTO
Quando **Object o Array** viene passato come prop, vengono **SEMPRE PASSATI DA REFERENCE** , il che significa indipendentemente dal tipo di binding esplicitamente definito `:email.sync="email"` o `:email="email"` o `:email.once="email"` , se l'e-mail è un oggetto o una matrice nel genitore, indipendentemente dal tipo di bind, qualsiasi modifica nel valore dell'elemento di prova all'interno del componente figlio influirà anche sul valore del genitore.

Puntelli come matrice

Nel file `contact-details.js` abbiamo definito `props:['phone', 'email']` come una matrice, che va bene se non vogliamo un controllo a grana fine con oggetti di scena.

Oggetti di scena come oggetto

Se vogliamo più controllo a grana fine su oggetti di scena, come

- se vogliamo definire quale tipo di valori sono accettabili come il prop
- quale dovrebbe essere un valore predefinito per il puntello
- se un valore DEVE (richiesto) essere passato per il prop o è opzionale

quindi abbiamo bisogno di usare la notazione degli oggetti per definire gli oggetti di scena, come abbiamo fatto in `address.js` .

Se stiamo creando componenti riutilizzabili che possono essere utilizzati anche da altri sviluppatori nel team, è buona pratica definire gli oggetti di scena come oggetti in modo che chiunque usi il componente abbia un'idea chiara di quale dovrebbe essere il tipo di dati e se è obbligatorio o facoltativo.

Si riferisce anche alla ***convalida degli oggetti*** di ***scena*** . Il **tipo** può essere uno qualsiasi dei seguenti costruttori nativi:

- Stringa
- Numero
- booleano
- schieramento
- Oggetto
- Funzione
- o un costruttore personalizzato

Alcuni esempi di convalida di oggetti di scena presi da http://vuejs.org/guide/components.html#Props

```
Vue.component('example', {
   props: {
       // basic type check (`null` means accept any type)
       propA: Number,
       // multiple possible types (1.0.21+)
       propM: [String, Number],
       // a required string
       propB: {
         type: String,
         required: true
       },
       // a number with default value
       propC: {
          type: Number,
          default: 100
       },
       // object/array defaults should be returned from a
       // factory function
```

```
       propD: {
          type: Object,
          default: function () {
             return { msg: 'hello' }
         }
       },
       // indicate this prop expects a two-way binding. will
       // raise a warning if binding type does not match.
       propE: {
          twoWay: true
       },
       // custom validator function
       propF: {
          validator: function (value) {
             return value > 10
          }
       },
       // coerce function (new in 1.0.12)
       // cast the value before setting it on the component
       propG: {
          coerce: function (val) {
            return val + '' // cast the value to string
          }
       },
       propH: {
          coerce: function (val) {
            return JSON.parse(val) // cast the value to Object
          }
       }
    }
});
```

CamelCase vs kebab-case

Gli attributi HTML non fanno distinzione tra `addresstype` e minuscole, il che significa che non è possibile distinguere tra tipo di `addresstype` e tipo di `addressType` , quindi quando si utilizzano i nomi di oggetti camelCase come attributi, è necessario utilizzare gli equivalenti di caso kebab (segno delimitato da delimitazione):
`addressType` dovrebbe essere scritto come `address-type` nell'attributo HTML.

Puntelli dinamici

Così come sei in grado di associare dati da una vista al modello, puoi anche legare oggetti di scena usando la stessa direttiva v-bind per passare informazioni dai componenti padre a figlio.

JS

```
new Vue({
    el: '#example',
    data: {
        msg: 'hello world'
    }
});

Vue.component('child', {
```

```
    props: ['myMessage'],
    template: '<span>{{ myMessage }}</span>
});
```

HTML

```
<div id="example">
    <input v-model="msg" />
    <child v-bind:my-message="msg"></child>
    <!-- Shorthand ... <child :my-message="msg"></child> -->
</div>
```

Risultato

```
hello world
```

Passando puntelli durante l'utilizzo di Vue JSX

Abbiamo un componente principale: importando un componente figlio in esso passeremo gli oggetti di scena tramite un attributo. Qui l'attributo è 'src' e stiamo passando anche 'src'.

ParentComponent.js

```
import ChildComponent from './ChildComponent';
export default {
    render(h, {props}) {
        const src = 'https://cdn-images-1.medium.com/max/800/1*AxRXW2j8qmGJixIYg7n6uw.jpeg';
        return (
           <ChildComponent src={src} />
        );
    }
};
```

E un componente figlio, dove abbiamo bisogno di passare oggetti di scena. Dobbiamo specificare quali oggetti stiamo passando.

ChildComponent.js:

```
export default {
    props: ['src'],
    render(h, {props}) {
        return (
            <a href = {props.src} download = "myimage" >
                Click this link
            </a>
        );
    }
```

```
};
```

Leggi puntelli online: https://riptutorial.com/it/vue-js/topic/3080/puntelli

Capitolo 20: Rendering condizionale

Sintassi

- `<element v-if="condition"></element>` // v-if
- `<element v-if="condition"></element><element v-else="condition"></element>` // v-if | v-else
- `<template v-if="condition">...</template>` // template v-if
- `<element v-show="condition"></element>` // v-show

Osservazioni

È molto importante ricordare la differenza tra `v-if` e `v-show` . Mentre i loro usi sono quasi identici, un elemento legato a `v-if` *verrà visualizzato solo nel DOM* quando la condizione è `true` per la **prima volta** . Quando si usa la direttiva `v-show` , *tutti gli elementi* ***sono*** *resi nel DOM* ma sono nascosti usando lo stile di `display` se la condizione è `false` !

Examples

Panoramica

In Vue.js, il rendering condizionale viene ottenuto utilizzando un insieme di direttive sugli elementi nel modello.

`v-if`

L'elemento viene visualizzato normalmente quando la condizione è `true` . Quando la condizione è `false` , si verifica solo *una* compilazione *parziale* e l'elemento non viene reso nel DOM finché la condizione non diventa `true` .

`v-else`

Non accetta una condizione, ma esegue il rendering dell'elemento se la condizione `v-if` dell'elemento precedente è `false` . Può essere utilizzato solo dopo un elemento con la direttiva `v-if` .

`v-show`

Si comporta in modo simile a `v-if` , tuttavia, l'elemento sarà *sempre* reso nel DOM, anche quando la condizione è `false` . Se la condizione è `false` , questa direttiva semplicemente imposta lo stile di `display` dell'elemento su `none` .

v-if / v-else

Supponendo che abbiamo un'istanza `Vue.js` definita come:

```
var vm = new Vue({
    el: '#example',
    data: {
        a: true,
        b: false
    }
});
```

Puoi rendere condizionalmente qualsiasi elemento html includendo la direttiva v-if; l'elemento che contiene v-if eseguirà il rendering solo se la condizione restituisce true:

```
<!-- will render 'The condition is true' into the DOM -->
<div id="example">
    <h1 v-if="a">The condition is true</h1>
</div>
```

L'elemento `<h1>` verrà visualizzato in questo caso, perché la variabile 'a' è vera. v-if può essere utilizzato con qualsiasi espressione, proprietà calcolata o funzione che restituisce un valore booleano:

```
<div v-if="0 === 1">                  false; won't render</div>
<div v-if="typeof(5) === 'number'">   true; will render</div>
```

È possibile utilizzare un elemento `template` per raggruppare più elementi insieme per una singola condizione:

```
<!-- in this case, nothing will be rendered except for the containing 'div' -->
<div id="example">
    <template v-if="b">
        <h1>Heading</h1>
        <p>Paragraph 1</p>
        <p>Paragraph 2</p>
    </template>
</div>
```

Quando si usa `v-if` , si ha anche la possibilità di integrare una condizione di contatore con la direttiva `v-else` . Il contenuto contenuto all'interno dell'elemento verrà visualizzato solo se la condizione della precedente v-if è falsa. Nota che questo significa che un elemento con v-else deve apparire immediatamente dopo un elemento con v-if.

```
<!-- will render only 'ELSE' -->
<div id="example">
    <h1 v-if="b">IF</h1>
    <h1 v-else="a">ELSE</h1>
</div>
```

Proprio come con v-if, con v-else puoi raggruppare più elementi html all'interno di un `<template>` :

```
<div v-if="'a' === 'b'"> This will never be rendered. </div>
<template v-else>
    <ul>
      <li> You can also use templates with v-else. </li>
```

```
      <li> All of the content within the template </li>
      <li> will be rendered. </li>
    </ul>
</template>
```

v-mostra

L'uso della direttiva `v-show` è quasi identico a quello di `v-if` . Le uniche differenze sono che `v-show` *non* supporta la sintassi `<template>` e non esiste una condizione "alternativa".

```
var vm = new Vue({
    el: '#example',
    data: {
        a: true
    }
});
```

L'uso di base è il seguente ...

```
<!-- will render 'Condition met' -->
<div id="example">
    <h1 v-show="a">Condition met</h1>
</div>
```

Mentre `v-show` non supporta la direttiva `v-else` per definire condizioni "alternative", questo può essere ottenuto negando il precedente ...

```
<!-- will render 'This is shown' -->
<div id="example">
    <h1 v-show="!a">This is hidden</h1>
    <h1 v-show="a">This is shown</h1>
</div>
```

Leggi Rendering condizionale online: https://riptutorial.com/it/vue-js/topic/3465/rendering-condizionale

Capitolo 21: slot

Osservazioni

Importante! Gli slot dopo il rendering non garantiscono l'ordine delle posizioni per le slot. Lo slot, che era il primo, potrebbe avere una posizione diversa dopo il rendering.

Examples

Utilizzando le singole slot

Gli slot singoli vengono utilizzati quando un componente figlio definisce solo uno `slot` nel modello. Il componente della `page` alto utilizza un singolo slot per distribuire il contenuto.

Un esempio della `page` modello del componente utilizzando un singolo slot è qui sotto:

```
<html>
    <head>
        <title>Page Title</title>
    </head>
    <body>
        <slot>
            This will only be displayed if there is no content
            to be distributed.
        </slot>
    </body>
</html>
```

Per illustrare come funziona lo slot, possiamo impostare una pagina come segue.

```
<page>
    <p>This content will be displayed within the page component</p>
</page>
```

Il risultato finale sarà:

```
<html>
    <head>
        <title>Page Title</title>
    </head>
    <body>
        <p>This content will be displayed within the page component</p>
    </body>
</html>
```

Se non avessimo inserito nulla tra i tag della `page` e avessimo `<page></page>` avremmo invece ottenuto il seguente risultato poiché esiste un contenuto predefinito tra i tag dello `slot` nel modello del componente della `page` .

```
<html>
    <head>
        <title>Page Title</title>
    </head>
    <body>
        This will only be displayed if there is no content
        to be distributed.
    </body>
</html>
```

Quali sono le slot?

Gli slot offrono un modo conveniente di distribuire il contenuto da un componente principale a un componente figlio. Questo contenuto può essere qualsiasi cosa, da testo, HTML o anche altri componenti.

A volte può essere utile pensare alle slot come mezzo per iniettare il contenuto direttamente nel modello di un componente figlio.

Gli slot sono particolarmente utili quando la composizione del componente sotto il componente principale non è sempre la stessa.

Prendi il seguente esempio in cui abbiamo un componente di `page` . Il contenuto della pagina potrebbe cambiare in base alla visualizzazione di tale pagina, ad esempio un articolo, post di blog o modulo.

Articolo

```
<page>
    <article></article>
    <comments></comments>
</page>
```

Post sul blog

```
<page>
    <blog-post></blog-post>
    <comments></comments>
</page>
```

Modulo

```
<page>
    <form></form>
</page>
```

Si noti come il contenuto del componente della `page` può cambiare. Se non usassimo gli slot questo sarebbe più difficile in quanto la parte interna del modello sarebbe stata riparata.

Ricorda: *"Tutto nel modello principale è compilato nell'ambito genitore, tutto nel modello figlio è compilato in ambito figlio".*

Uso di slot con nome

Gli slot con nome funzionano in modo simile alle singole slot, ma consentono di distribuire il contenuto in aree diverse all'interno del modello di componente figlio.

Prendi il componente della `page` dall'esempio precedente ma modifica il suo modello in modo che sia come segue:

```
<html>
    <head>
        <title>Page Title</title>
    </head>
    <body>
        <aside>
            <slot name="sidebar"></slot>
        </aside>
        <main>
            <slot name="content"></slot>
        </main>
    </body>
</html>
```

Quando si utilizza il componente della `page` , ora possiamo determinare dove viene posizionato il contenuto tramite l'attributo dello `slot` :

```
<page>
    <p slot="sidebar">This is sidebar content.</p>
    <article slot="content"></article>
</page>
```

La pagina risultante sarà:

```
<html>
    <head>
        <title>Page Title</title>
    </head>
    <body>
        <aside>
            <p>This is sidebar content.</p>
        </aside>
        <main>
            <article></article>
        </main>
    </body>
</html>
```

Se uno `slot` è definito senza un attributo `name` qualsiasi contenuto che viene collocato all'interno dei tag del componente che non specifica un attributo `slot` verrà inserito nello slot.

Vedi l'esempio di multi inserzione sui documenti ufficiali di Vue.js.

Usare gli slot in Vue JSX con 'babel-plugin-transform-vue-jsx'

Se stai usando VueJS2 e ti piace usare JSX insieme ad esso. In questo caso, per usare lo slot, la soluzione con l'esempio è sotto. Dobbiamo usare `this.$slots.default` È quasi come `this.props.children` in React JS.

Component.js:

```
export default {
    render(h) {  //eslint-disable-line
        return (
            <li>
                { this.$slots.default }
            </li>
        );
    }
};
```

ParentComponent.js

```
import Component from './Component';

export default {
    render(h) {  //eslint-disable-line
        return (
            <ul>
                <Component>
                    Hello World
                </Component>
            </ul>
        );
    }
};
```

Leggi slot online: https://riptutorial.com/it/vue-js/topic/4484/slot

Capitolo 22: Usando "questo" in Vue

introduzione

Uno degli errori più comuni che troviamo nel codice Vue su StackOverflow è l'uso improprio di `this` . Gli errori più comuni ricadono generalmente in due aree, utilizzando `this` in callback per promesse o altre funzioni asincrone e utilizzando le funzioni freccia per definire metodi, proprietà calcolate, ecc.

Examples

SBAGLIATO! Usando "questo" in un callback all'interno di un metodo Vue.

```
new Vue({
  el:"#app",
  data:{
    foo: "bar"
  },
  methods:{
    doSomethingAsynchronous(){
      setTimeout(function(){
        // This is wrong! Inside this function,
        // "this" refers to the window object.
        this.foo = "baz";
      }, 1000);
    }
  }
})
```

SBAGLIATO! Usare "questo" all'interno di una promessa.

```
new Vue({
  el:"#star-wars-people",
  data:{
    people: null
  },
  mounted: function(){
    $.getJSON("http://swapi.co/api/people/", function(data){
      // Again, this is wrong! "this", here, refers to the window.
      this.people = data.results;
    })
  }
})
```

DESTRA! Usa una chiusura per catturare "questo"

Puoi catturare `this` corretto usando una chiusura .

```
new Vue({
  el:"#star-wars-people",
```

```
  data:{
    people: null
  },
  mounted: function(){
    // Before executing the web service call, save this to a local variable
    var self = this;
    $.getJSON("http://swapi.co/api/people/", function(data){
      // Inside this call back, because of the closure, self will
      // be accessible and refers to the Vue object.
      self.people = data.results;
    })
  }
})
```

DESTRA! Usa il bind.

È possibile associare la funzione di callback.

```
new Vue({
  el:"#star-wars-people",
  data:{
    people: null
  },
  mounted:function(){
    $.getJSON("http://swapi.co/api/people/", function(data){
      this.people = data.results;
    }.bind(this));
  }
})
```

DESTRA! Usa una funzione freccia.

```
new Vue({
  el:"#star-wars-people",
  data:{
    people: null
  },
  mounted: function(){
    $.getJSON("http://swapi.co/api/people/", data => this.people = data.results);
  }
})
```

Attenzione! Le funzioni freccia sono una sintassi introdotta in Ecmascript 2015. Non è ancora supportata ma *tutti* i browser moderni, quindi usala solo se stai mirando a un browser che *conosci* , oppure se stai compilando il tuo javascript con la sintassi ES5 usando qualcosa come babel .

SBAGLIATO! Utilizzando una funzione freccia per definire un metodo che si riferisce a "questo"

```
new Vue({
  el:"#app",
  data:{
    foo: "bar"
  },
```

```
  methods:{
    // This is wrong! Arrow functions capture "this" lexically
    // and "this" will refer to the window.
    doSomething: () => this.foo = "baz"
  }
})
```

DESTRA! Definire metodi con la sintassi della funzione tipica

```
new Vue({
  el:"#app",
  data:{
    foo: "bar"
  },
  methods:{
    doSomething: function(){
      this.foo = "baz"
    }
  }
})
```

In alternativa, se stai usando un compilatore javascript o un browser che supporta Ecmascript 2015

```
new Vue({
  el:"#app",
  data:{
    foo: "bar"
  },
  methods:{
    doSomething(){
      this.foo = "baz"
    }
  }
})
```

Leggi Usando "questo" in Vue online: https://riptutorial.com/it/vue-js/topic/9350/usando--questo--in-vue

Capitolo 23: Vue singoli componenti di file

introduzione

Descrivere come creare singoli componenti di file in un file .vue.

Specialmente le decisioni di progettazione che possono essere prese.

Examples

Esempio di file componente .vue

```
<template>
  <div class="nice">Component {{title}}</div>
</template>

<script>
export default {
    data() {
        return {
            title: "awesome!"
        };
    }
}
</script>

<style>
.nice {
    background-color: red;
    font-size: 48px;
}
</style>
```

Leggi Vue singoli componenti di file online: https://riptutorial.com/it/vue-js/topic/10118/vue-singoli-componenti-di-file

Capitolo 24: VueJS + Redux con Vua-Redux (soluzione migliore)

Examples

Come usare Vua-Redux

Installazione di Vua Redux da NPM:

Installa attraverso:

```
npm i vua-redux --save
```

Inizializzare:

==========

// main.js

```
import Vue from 'vue';
import { reduxStorePlugin } from 'vua-redux';
import AppStore from './AppStore';
import App from './Component/App';

// install vua-redux
Vue.use(reduxStorePlugin);

new Vue({
    store: AppStore,
    render(h) {
        return <App />
    }
});
```

// AppStore.js

```
import { createStore } from 'redux';

const initialState = {
  todos: []
};

const reducer = (state = initialState, action) => {
  switch(action.type){
    case 'ADD_TODO':
      return {
        ...state,
        todos: [...state.todos, action.data.todo]
      }
```

```
    default:
      return state;
    }
}

const AppStore = createStore(reducer);

export default AppStore;
```

Usa nel tuo componente:

// components / App.js

```
import { connect } from 'vua-redux';

const App = {
    props: ['some-prop', 'another-prop'],

    /**
     * everything you do with vue component props
     * you can do inside collect key
     */
    collect: {
        todos: {
            type: Array,
        },
        addTodo: {
            type: Function,
        },
    },

    methods: {
        handleAddTodo() {
            const todo = this.$refs.input.value;
            this.addTodo(todo);
        }
    },

    render(h) {
        return <div>
            <ul>
                {this.todos.map(todo => <li>{todo}</li>)}
            </ul>

            <div>
                <input type="text" ref="input" />
                <button on-click={this.handleAddTodo}>add todo</button>
            </div>
        </div>
    }
};

function mapStateAsProps(state) {
    return {
        todos: state.todos
    };
}

function mapActionsAsProps(dispatch) {
    return {
```

```
        addTodo(todo) {
            dispatch({
                type: 'ADD_TODO',
                data: { todo }
            })
        }
    }
}

export default connect(mapStateAsProps, mapActionsAsProps)(App);
```

Leggi VueJS + Redux con Vua-Redux (soluzione migliore) online: https://riptutorial.com/it/vue-js/topic/7396/vuejs-plus-redux-con-vua-redux--soluzione-migliore-

Capitolo 25: vue-router

introduzione

vue-router è la libreria di routing ufficialmente supportata per vue.js.

Sintassi

- `<router-link to="/path">Link Text</router-link> <!-- Creates a link to the route that matches the path -->`
- `<router-view></router-view> <!-- Outlet for the currently matched route. It's component will be rendered here. -->`

Examples

Routing di base

Il modo più semplice per iniziare e utilizzare con vue-router è utilizzare la versione fornita tramite CDN.

HTML:

```
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script src="https://unpkg.com/vue-router/dist/vue-router.js"></script>

<div id="router-example">
    <router-link to="/foo">Link to Foo route</router-link>
    <router-view></router-view>
</div>
```

JavaScript (ES2015):

```
const Foo = { template: <div>This is the component for the Foo route</div> }

const router = new VueRouter({
    routes: [
       { path: '/foo', component: Foo}
    ]
})

const routerExample = new Vue({
    router
}).$mount('#router-example')
```

Leggi vue-router online: https://riptutorial.com/it/vue-js/topic/9654/vue-router

Capitolo 26: Vuex

introduzione

Vuex è una libreria di gestione dello stato + libreria per le applicazioni Vue.js. Serve da archivio centralizzato per tutti i componenti di un'applicazione, con regole che garantiscono che lo stato possa essere mutato solo in modo prevedibile. Si integra inoltre con l'estensione ufficiale degli strumenti di sviluppo di Vue per fornire funzionalità avanzate quali il debugging della corsa a zero zero e l'esportazione / importazione dello snapshot di stato.

Examples

Cos'è Vuex?

Vuex è un plug-in ufficiale per Vue.js che offre un archivio dati centralizzato da utilizzare all'interno dell'applicazione. È fortemente influenzato dall'architettura dell'applicazione Flux che presenta un flusso di dati unidirezionale che porta a una progettazione e un ragionamento delle applicazioni più semplici.

All'interno di un'applicazione Vuex, il datastore contiene tutti gli stati di **applicazione condivisi** . Questo stato è alterato dalle **mutazioni** che vengono eseguite in risposta a **un'azione che** richiama un evento di mutazione tramite il **dispatcher** .

Un esempio del flusso di dati in un'applicazione Vuex è descritto nello schema seguente.

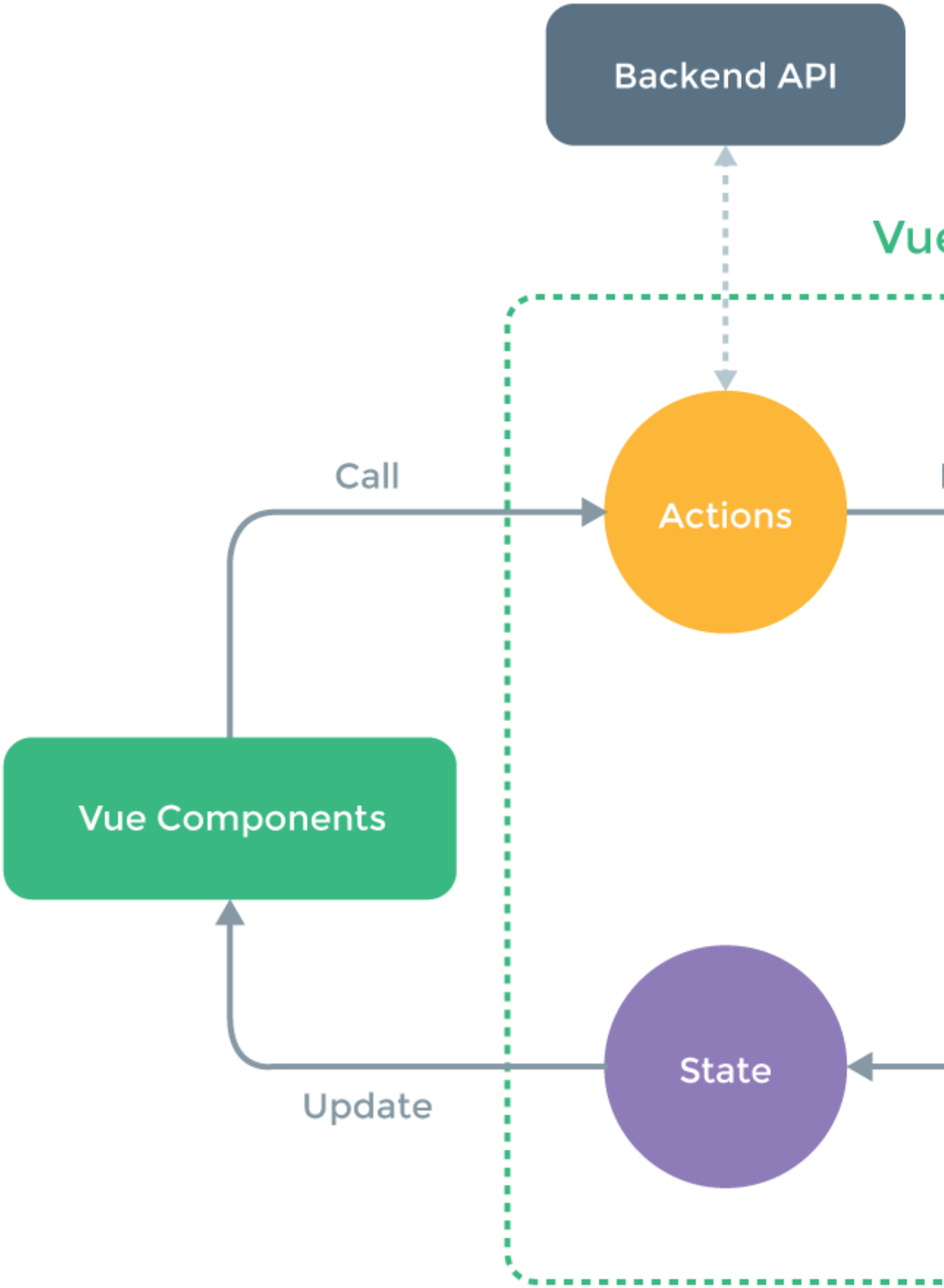

Backend API
Vue
Call
Actions
Vue Components
State
Update

Diagramma utilizzato con la licenza MIT , originariamente dal repository ufficiale Vuex GitHub .

Singoli componenti dell'applicazione Vue.js possono accedere all'oggetto negozio per recuperare i dati tramite **getter** , che sono funzioni pure che restituiscono una copia di sola lettura dei dati desiderati.

I componenti possono avere **azioni** che sono funzioni che eseguono modifiche alla propria copia dei dati del componente, quindi utilizzare il **dispatcher** per inviare un evento di mutazione. Questo evento viene quindi gestito dal datastore che aggiorna lo stato secondo necessità.

Le modifiche vengono quindi automaticamente riflesse in tutta l'applicazione poiché tutti i componenti sono legati in modo reattivo allo store tramite i loro getter.

Un esempio che illustra l'uso di vuex in un progetto vue.

```
const state = {
    lastClickTime: null
}

const mutations = {
    updateLastClickTime: (state, payload) => {
      state.lastClickTime = payload
  }
}

const getters = {
  getLastClickTime: state => {
    return new Date(state.lastClickTime)
  }
}

const actions = {
    syncUpdateTime: ({ commit }, payload) => {
    commit("updateLastClickTime", payload)
  },
  asyncUpdateTime: ({ commit }, payload) => {
    setTimeout(() => {
      commit("updateLastClickTime", payload)
    }, Math.random() * 5000)
  }
}

const store = new Vuex.Store({
  state,
  getters,
  mutations,
  actions
})

const { mapActions, mapGetters } = Vuex;

// Vue
const vm = new Vue({
    el: '#container',
  store,
  computed: {
      ...mapGetters([
```

```
        'getLastClickTime'
    ])
  },
  methods: {
      ...mapActions([
        'syncUpdateTime',
      'asyncUpdateTime'
    ]),
    updateTimeSyncTest () {
        this.syncUpdateTime(Date.now())
    },
    updateTimeAsyncTest () {
        this.asyncUpdateTime(Date.now())
    }
  }
})
```

E il modello HTML per lo stesso:

```
<div id="container">
  <p>{{ getLastClickTime || "No time selected yet" }}</p>
  <button @click="updateTimeSyncTest">Sync Action test</button>
  <button @click="updateTimeAsyncTest">Async Action test</button>
</div>
```

1. Qui lo **stato** contiene la proprietà **lastClickTime** inizializzata come null. Questa impostazione di valori predefiniti è importante per mantenere **reattive** le proprietà. **Le proprietà non menzionate nello stato** saranno disponibili ma le modifiche apportate in seguito **non sarebbero accessibili** utilizzando i getter.

2. Il getter utilizzato fornisce una proprietà calcolata che verrà aggiornata ogni volta che una mutazione aggiorna il valore della proprietà di stato.

3. **Solo le mutazioni** possono cambiare lo stato e le sue proprietà, ciò detto, lo fa **solo in** modo **sincrono** .

4. Un'Azione può essere usata in caso di aggiornamenti asincroni, dove la chiamata API (qui derisa dal setTimeout a tempo casuale) può essere fatta nell'azione, e dopo aver ottenuto la risposta a cui può essere impegnata una mutazione, per effettuare la modifica allo stato .

Perché usare Vuex?

Quando si costruiscono applicazioni di grandi dimensioni come le applicazioni per pagina singola (SPA), che in genere sono costituite da molti componenti riutilizzabili, possono diventare rapidamente difficili da costruire e gestire. Anche la condivisione di dati e stato tra questi componenti può essere rapidamente interrotta e diventa difficile eseguirne il debug e la manutenzione.

Utilizzando un archivio dati centralizzato dell'applicazione, l'intero stato dell'applicazione può essere rappresentato in un unico punto, rendendo l'applicazione più organizzata. Attraverso l'uso di un flusso di dati unidirezionale, le mutazioni e l'accesso ai dati del componente scoping solo ai dati richiesti, diventa molto più semplice ragionare sul ruolo del componente e su come dovrebbe

influire sullo stato dell'applicazione.

I componenti VueJS sono entità separate e non possono condividere facilmente dati tra loro. Per condividere i dati senza vuex, dobbiamo `emit` eventi con dati e quindi ascoltare e catturare quell'evento con `on` .

componente 1

```
this.$emit('eventWithDataObject', dataObject)
```

componente 2

```
this.$on('eventWithDataObject', function (dataObject) {
    console.log(dataObject)
})
```

Con vuex installato possiamo semplicemente accedere ai suoi dati da qualsiasi componente senza bisogno di ascoltare gli eventi.

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

Possiamo inoltre modificare i dati in modo sincrono con i *mutatori* , utilizzare *azioni* asincrone e ottenere dati con le funzioni *getter* .

Le funzioni Getter potrebbero funzionare come funzioni globali calcolate. Possiamo accedervi dai componenti:

```
this.$store.getters.myGetter
```

Le azioni sono metodi globali. Possiamo spedirli dai componenti:

```
this.$store.dispatch('myAction', myDataObject)
```

E le mutazioni sono l'unico modo per cambiare i dati in vuex. Siamo in grado di commettere modifiche:

```
this.$store.commit('myMutation', myDataObject)
```

Il codice Vuex sarebbe simile a questo

```
state: {
  myData: {
    key: 'val'
  }
},
getters: {
  myGetter: state => {
     return state.myData.key.length
  }
},
```

```
actions: {
  myAction ({ commit }, myDataObject) {
    setTimeout(() => {
      commit('myMutation', myDataObject)
    }, 2000)
  }
},
mutations: {
  myMutation (state, myDataObject) {
    state.myData = myDataObject
  }
}
```

Come installare Vuex?

La maggior parte delle volte che utilizzerai Vuex sarà in applicazioni basate su componenti più grandi, dove probabilmente utilizzerai un bundle di moduli come Webpack o Browserify insieme a Vueify se utilizzi file singoli.

In questo caso il modo più semplice per ottenere Vuex è da NPM. Eseguire il comando seguente per installare Vuex e salvarlo nelle dipendenze dell'applicazione.

```
 npm install --save vuex
```

Assicurati di caricare il link Vuex con il tuo setup Vue posizionando la seguente riga dopo la tua `require('vue')` .

```
Vue.use(require('vuex'))
```

Vuex è anche disponibile su CDN; si può afferrare l'ultima versione dal cdnjs qui .

Notifiche auto licenziabili

Questo esempio registra dinamicamente un modulo vuex per la memorizzazione di notifiche personalizzate che possono essere automaticamente eliminate

notifications.js

Risolvi vuex store e definisci alcune costanti

```
//Vuex store previously configured on other side
import _store from 'path/to/store';

//Notification default duration in milliseconds
const defaultDuration = 8000;

//Valid mutation names
const NOTIFICATION_ADDED = 'NOTIFICATION_ADDED';
const NOTIFICATION_DISMISSED = 'NOTIFICATION_DISMISSED';
```

imposta il nostro stato iniziale del modulo

```
const state = {
  Notifications: []
}
```

imposta i getter del modulo

```
const getters = {
  //All notifications, we are returning only the raw notification objects
  Notifications: state => state.Notifications.map(n => n.Raw)
}
```

imposta il nostro modulo Azioni

```
const actions = {
  //On actions we receive a context object which exposes the
  //same set of methods/properties on the store instance
  //{commit} is a shorthand for context.commit, this is an
  //ES2015 feature called argument destructuring
  Add({ commit }, notification) {
    //Get notification duration or use default duration
    let duration = notification.duration || defaultDuration

    //Create a timeout to dismiss notification
    var timeOut = setTimeout(function () {
      //On timeout mutate state to dismiss notification
      commit(NOTIFICATION_DISMISSED, notification);
    }, duration);

    //Mutate state to add new notification, we create a new object
    //for save original raw notification object and timeout reference
    commit(NOTIFICATION_ADDED, {
      Raw: notification,
      TimeOut: timeOut
    })
  },
  //Here we are using context object directly
  Dismiss(context, notification) {
    //Just pass payload
    context.commit(NOTIFICATION_DISMISSED, notification);
  }
}
```

imposta le nostre mutazioni di modulo

```
const mutations = {
  //On mutations we receive current state and a payload
  [NOTIFICATION_ADDED](state, notification) {
    state.Notifications.push(notification);
  },
  //remember, current state and payload
  [NOTIFICATION_DISMISSED](state, rawNotification) {
    var i = state.Notifications.map(n => n.Raw).indexOf(rawNotification);
    if (i == -1) {
      return;
    }

    clearTimeout(state.Notifications[i].TimeOut);
```

```
    state.Notifications.splice(i, 1);
  }
}
```

Registra il nostro modulo con stato, getter, azioni e mutazione definiti

```
_store.registerModule('notifications', {
  state,
  getters,
  actions,
  mutations
});
```

uso

componentA.vue

Questo componente visualizza tutte le notifiche come avvisi di bootstrap nell'angolo in alto a destra dello schermo, inoltre consente di ignorare manualmente ciascuna notifica.

```
<template>
<transition-group name="notification-list" tag="div" class="top-right">
  <div v-for="alert in alerts" v-bind:key="alert" class="notification alert alert-dismissible"
v-bind:class="'alert-'+alert.type">
    <button v-on:click="dismiss(alert)" type="button" class="close" aria-label="Close"><span
aria-hidden="true">&times;</span></button>
    <div>
      <div>
        <strong>{{alert.title}}</strong>
      </div>
      <div>
        {{alert.text}}
      </div>
    </div>
  </div>
</transition-group>
</template>

<script>
export default {
  name: 'arc-notifications',
  computed: {
    alerts() {
      //Get all notifications from store
      return this.$store.getters.Notifications;
    }
  },
  methods: {
    //Manually dismiss a notification
    dismiss(alert) {
      this.$store.dispatch('Dismiss', alert);
    }
  }
}
</script>
<style lang="scss" scoped>
$margin: 15px;
```

```
.top-right {
    top: $margin;
    right: $margin;
    left: auto;
    width: 300px;
    //height: 600px;
    position: absolute;
    opacity: 0.95;
    z-index: 100;
    display: flex;
    flex-wrap: wrap;
    //background-color: red;
}
.notification {
    transition: all 0.8s;
    display: flex;
    width: 100%;
    position: relative;
    margin-bottom: 10px;
    .close {
        position: absolute;
        right: 10px;
        top: 5px;
    }

    > div {
        position: relative;
        display: inline;
    }
}
.notification:last-child {
    margin-bottom: 0;
}
.notification-list-enter,
.notification-list-leave-active {
    opacity: 0;
    transform: translateX(-90px);
}
.notification-list-leave-active {
    position: absolute;
}
</style>
```

Snippet per aggiungere notifiche in qualsiasi altro componente

```
//payload could be anything, this example content matches with componentA.vue
this.$store.dispatch('Add', {
  title = 'Hello',
  text = 'World',
  type = 'info',
  duration = 15000
});
```

Leggi Vuex online: https://riptutorial.com/it/vue-js/topic/3430/vuex

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con Vue.js	Community, Erick Petrucelli, ironcladgeek, J. Bruni, James, Lambda Ninja, m_callens, MotKohn, rap-2-h, Ru Chern Chong, Sankalp Singha, Shog9, Shuvo Habib, user1012181, user6939352, Yerko Palma
2	Associazione dati	gurghet, Jilson Thomas
3	Bus degli eventi	Amresh Venugopal
4	componenti	Donkarnash, Elfayer, Hector Lorenzo, Jeff, m_callens, phaberest, RedRiderX, user6939352
5	Componenti dinamici	Med, Ru Chern Chong
6	Componenti personalizzati con v-model	Amresh Venugopal
7	Direttive personalizzate	Mat J, Ogie Sado
8	Elenco di rendering	chuanxd, gurghet, Mahmoud, Theo
9	eventi	Elfayer
10	Filtri personalizzati	Finrod, M U, m_callens
11	L'array modifica le avvertenze sul rilevamento	Vamsi Krishna
12	Lifecycle Hooks	Linus Borg, m_callens, PatrickSteele, xtreak
13	mixins	Ogie Sado
14	Modello "webpack" di Polyfill	Stefano Nepa
15	modificatori	sept08
16	osservatori	El_Matella
17	plugin	AldoRomo88

18	Proprietà calcolate	Amresh Venugopal, cl3m, jaredsk, m_callens, Theo, Yerko Palma
19	puntelli	asemahle, Donkarnash, FlatLander, m_callens, rap-2-h, Shuvo Habib
20	Rendering condizionale	jaredsk, m_callens, Nirazul, user6939352
21	slot	Daniel Waghorn, Elfayer, Shuvo Habib, Slava
22	Usando "questo" in Vue	Bert
23	Vue singoli componenti di file	jordiburgos, Ru Chern Chong
24	VueJS + Redux con Vua-Redux (soluzione migliore)	Aniko Litvanyi, FlatLander, Shuvo Habib, Stefano Nepa
25	vue-router	AJ Gregory
26	Vuex	AldoRomo88, Amresh Venugopal, Daniel Waghorn, Matej Vrzala M4, Ru Chern Chong