# KIẾN THỨC CƠ BẢN

## Vue là gì

Là một framework Javascript xây dựng giao diện người dung, xây dựng dựa trên html, css, javascript

## Single File Components

Trong hầu hết các dự án vue xây dựng, sẽ tạo các thành phần Vue bằng cách sử dụng các định dạng giống html được gọi là SFC

## Cấu trúc cơ bản của .vue

<script setup>

// viết mã javascript tại đây

</script>

<template>

// viết mã html tại đây

</template>

<style scope>

// Viết mã css tại đây

</style>

* script: viết mã javascript hoặc typescript (lang = "ts") tại đây, "setup" là cú pháp tích hợp hàm khởi tạo chỉ có ở bản vue 3
* template: nơi viết mã html và các biến sử dụng trong ngoặc {{ }}
* style: nơi viết mã css, "scope" sẽ giúp việc tạo các biến css độc lập duy nhất giữa các file khác tránh dính css lẫn nhau

## API

Có 2 kiểu API sử dụng trong vue là: Option API và Composition API, trong tài liệu này sẽ sử dụng Composition API

* Option API: thường sử dụng cho Vue 2 chia các phần thành: data, methods, computed, watch, lifecycle hooks thành các khối riêng biệt. Tuy nhiên đối với dự án lớn nó trở nên phức tạp và phân tán, khó kiểm soát và chia sẻ giữa các component
* Composition API: sử dụng cho Vue 3, linh hoạt hiện đại, tổ chức và gom nhóm đơn giản hơn, tái sử dụng phù hợp cho typescript

## Tạo một ứng dụng Vue

Yêu cầu: Nodejs ít nhất 18.3 hoặc cao hơn

Tại thư mục làm việc (chưa tạo folder dự án)

npm create vue@latest

Nhập tên của dự án

Có thêm TypeScript không? (tùy chọn)

JSX (không)

Router (có)

Pinia (nếu muốn dùng để lưu trữ các state <trạng thái> - có)

Vitest (không)

ESLint (có)

Prettier (có)

Add Vue DevTools 7 (không)

cd <your-project-name>

npm install

npm run dev

Nếu ứng dụng đã sẵn sàng mọi thứ và đưa vào môi trường product

npm run build

## Cấu trúc thư mục cơ bản

src/

- components: nơi lưu trữ các SFC dùng chung

- composables: nơi lưu trữ các script dùng chung

- enums: nơi lưu trữ các hằng số

- layouts: nơi lưu trữ các layout

- plugins: pinia, axios, casl,...

- router: nơi lưu trữ là xử lý router

- stores: nơi lưu trữ các bộ nhớ dùng chung

- types: các prototype của class, object, ... sẽ được lưu tại đây

- views: các page sẽ được lưu tại đây

# CÁC YẾU TỐ CẦN THIẾT

## Tạo một ứng dụng vue

Mọi ứng dụng vue đều bắt đầu bằng hàm createApp

import { createApp } from 'vue'

const app = createApp({

/\* root component options \*/

})

## Cú pháp template

Đối với các biến khi in ra các thẻ html sẽ có dạng là văn bản, nên dù văn bản đó có chứa mã html thì cũng sẽ không được áp dụng, để khắc phục điều này ta có thuộc tính v-html. Tuy nhiên cần lưu ý khi sử dụng v-html vì dẫn đến XSS, và chỉ sử dụng khi nội dung đáng tin cậy

VD:

<script setup>

  const raw = '<p style="color: red">Hello world</p>';

</script>

<template>

  <p v-html="raw"></p>

</template>

### Thuộc tính Binding

Đối với những giá trị của thuộc tính có thể thay đổi được (dữ liệu động) thì phải dùng binding để khi giá trị của thuộc tính thay đổi thì thẻ đó cũng thay đổi theo

VD:

id => v-bind:id

class => v-bind:class

src = v-bind:src

Có thể viết rút gọn là:

:id

:class

:src

Lưu ý chỉ sử dụng v-bind cho thuộc tính động/ biến, còn đối với giá trị tĩnh thì vẫn dùng thuộc tính của html

cú pháp: v-bind:

VD:

<script setup>

  const imageUrl = 'https://upload.wikimedia.org/wikipedia/commons/thumb/9/95/Vue.js\_Logo\_2.svg/1184px-Vue.js\_Logo\_2.svg.png';

</script>

<template>

  <img v-bind:src="imageUrl" alt="">

</template>

Nếu bạn có nhiều thuộc tính cho 1 thẻ bạn có thể viết dạng object

const objectOfAttrs = {

id: 'container',

class: 'wrapper',

style: 'background-color:green'

}

<div v-bind="objectOfAttrs"></div>

### v-if

Thêm vào DOM nếu là true, và xóa khỏi DOM nếu là false

<button v-if="true">Click me</button>

### v-show

Hiển thị nếu là true, ẩn đi nếu là false (khác với v-if là v-show chỉ ẩn đi display: none chứ không hoàn toàn biến mất)

### v-for

Vòng lặp

VD đơn giản:

<script setup>

  const arr = [1,2,3];

</script>

<template>

  <ul>

    <li v-for="item in arr">{{ item }}</li>

  </ul>

</template>

VD đầy đủ:

<script setup>

  const obj = {

    'name': 'Alice',

    'age': 30,

    'city': 'New York'

  }

</script>

<template>

  <ul>

    <li v-for="(value, key, index) in obj" :key="key">

      {{ index }} - {{ key }} - {{ value }}

    </li>

  </ul>

</template>

### v-slot

### v-on

### Đối số động

Các thuộc tính có thể lựa chọn làm đối số thay đổi

VD:

<a v-bind:[attributeName]="url"> ... </a>

attributeName có thể là src, class, id hoàn toàn có thể thay đổi

Rút gọn:

<a :[attributeName]="url"> ... </a>

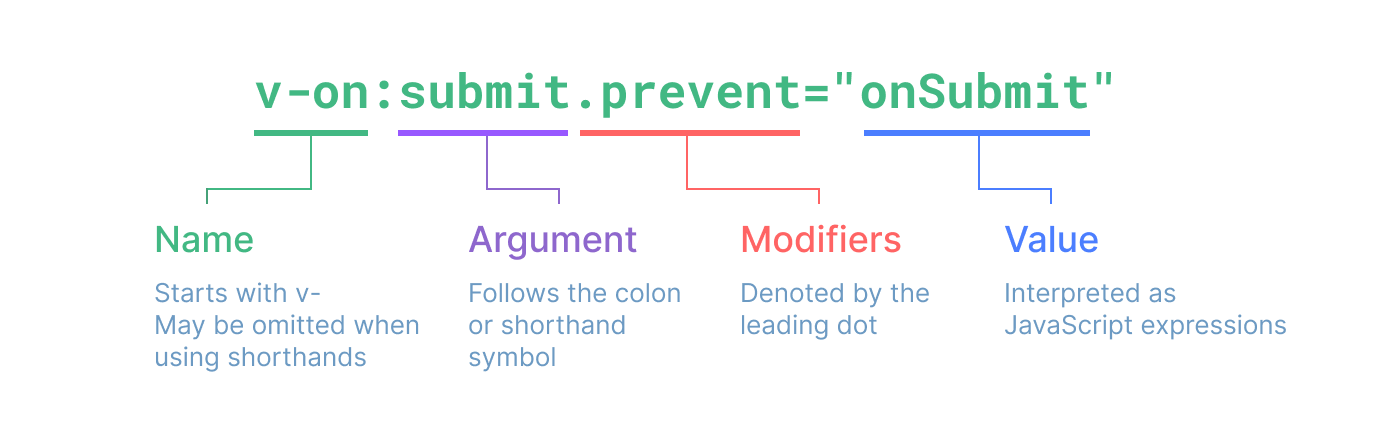
### Modifiers

Modifiers là loại hậu tố đặc biệt và chỉ được truy cập bằng dấu chấm

VD: khi submit 1 form sẽ có thao tác mặc định là load lại trang hoặc chuyển trang, để ngăn điều này trong v-on:click có loại hậu tố là prevent (preventDefault)

<button v-on:click.prevent>Click me</button>

Như vậy thì sẽ không còn load trang nữa



## Quản lý State

Khi làm việc với vue, bạn sẽ có những thao tác làm thay đổi giá trị của biến, và nếu gán cho biến mặc định của js thì sẽ không có thay đổi nào được hiển thị ra màn hình, do đó state ra đời để theo dõi sự thay đổi của các biến này từ đó thay hiển thị ra màn hình kết quả tương ứng

Có 2 loại quản lý State là: ref và reactive

### Ref

<script setup>

  import { ref } from 'vue'

  const count = ref(0)

</script>

<template>

  <button @click="count++">Count is: {{ count }}</button>

</template>

Ví dụ trên hiển thị ra nút bộ đếm, sẽ hiển thị lại số sau mỗi lần nhấn và số này sẽ tăng dần 1 đơn vị

Lưu ý: khi sử dụng giá trị của ref trong script thì cần truy cập hậu tố .value, còn trong template thì sử dụng trực tiếp không cần hậu tố

Ví dụ muốn lấy hoặc sửa giá trị của count trong script thì sẽ là: count.value

Giá trị của ref có thể là bất cứ thứ gì kể cả mảng hay đối tượng

Tuy nhiên nhược điểm của ref là không thể phát hiện ra giá trị trong mảng hay đối tượng thay đổi, nó chỉ phát hiện ra khi bạn thay đổi cả mảng (gán mảng mới), do đó ref chỉ phù hợp với dữ liệu đơn giản

### Reactive

Phù hợp với dữ liệu phức tạp như mảng hoặc đối tượng, có thể theo dõi sâu trong từng phần tử đó

Đối với reative, không cần truy cập giá trị thông qua .value

reactive không so sánh với đối tượng ban đầu như ref, mà reactive sử dụng proxy của JS để bao bọc đối tượng để lắng nghe các thay đổi trên nó, vậy nên khi so sánh nó sẽ so sánh proxy ban đầu chứ không so sánh đối tượng ban đầu

Proxy là đối tượng đại diện cho 1 đối tượng, theo dõi sự thay đổi trong trạng thái của đối tượng

VD:

const raw = {}

const proxy = reactive(raw)

console.log(proxy === raw) // false

Qua ví dụ trên có thể thấy mặc dù 2 đối tượng cùng khởi tạo như nhau nhưng khác nhau về proxy nên không thể bằng nhau (bạn A có 2 quả táo, bạn B có 2 quả táo nhưng thực chất táo của 2 bạn là khác nhau không phải là 1)

VD1

const raw = {};

const proxy = reactive(raw)

console.log(raw === proxy) // false

raw là đối tượng gốc tham chiếu đến vùng nhớ X

proxy là đối tượng proxy bao bọc raw tham chiếu đến vùng nhớ Y

nên chúng khác nhau

VD2

const raw = {};

const proxy = reactive(raw)

console.log(reactive(raw) === proxy) // true

console.log(reactive(proxy) === proxy) // true

reactive(raw) được lớp proxy bao bọc raw tham chiếu vùng nhớ X

reactive(proxy) được lớp proxy bao bọc lớp reactive(raw):

reactive(reactive(raw)) = reactive(raw) và sẽ không sinh ra vùng nhớ mới mà sẽ lấy luôn của react(raw) điều này là vì chúng được lưu cùng weakmap nên tham chiếu đến vùng nhớ X

Do đó chúng đều là true

VD3

const raw = reactive({})

const proxy = reactive({})

console.log(raw === proxy) // false

Mặc dù cả 2 đều tạo mới lớp proxy rỗng, tuy nhiên chúng được tham chiếu đến vùng nhớ khác nhau nên thực chất chúng khác nhau

Điều đặc biệt của reactive là khi thay đổi giá trị, thuộc tính trap 'set' đã được kích hoạt và thông báo cho vue cập nhật DOM chứ không phải so sánh proxy

Hạn chế:

* Chỉ hoạt động với dữ liệu phức tạp: mảng, đối tượng, map, set. Không hoạt động với dữ liệu nguyên thủy
* Không thể thay đổi toàn bộ đối tượng, vì chúng hoạt động dựa trên quyền truy cập thuộc tính đối tượng, nên luôn phải giữ kết nối đến cùng 1 tham chiếu trên đối tượng đó, nên không thể dễ dàng thay đổi
* Không hoạt động với phân giải cấu trúc (destructure), vì khi destructure là sẽ nhận 1 bản sao của thuộc tính đối tượng đó, không phải đối tượng thực sự nên không hoạt động reactive

### Sự kết hợp giữa ref và reactive

VD:  
const count = ref(0)

const state = reactive({

count

})

console.log(state.count) // 0

state.count = 1

console.log(count.value) // 1

* Qua ví dụ trên thấy được rằng khi thay đổi count thì cả ref và reactive đều ảnh hưởng
* Nhưng nếu thay thế count của reactive bằng ref khác thì mối liên kết cũ sẽ bị thay thế bởi ref mới
* Và sự kết hợp này chỉ hoạt động khi reactive là thuộc tính sâu, nếu là nông thì sẽ không hoạt động mà nó sẽ sinh ra một thuộc tính độc lập mới nhưng chỉ cùng tên với ref

import { ref, reactive } from 'vue';

const count = ref(0);

const state = reactive({ count });

console.log(state.count); // 0

state.count = 1; // Thay đổi ở đây sẽ không ảnh hưởng đến count.value

console.log(count.value); // Vẫn là 0

### nextTick()

Thông thường, khi bạn thay đổi giá trị cho dù bao nhiêu lần đi chăng nữa thì kết thúc lần thay đổi đó thì vue mới render lại phần tử liên quan đến giá trị thay đổi đó, dữ liệu chưa được thay đổi xong sẽ được lưu vào bộ đệm

VD:

count++;

count++;

count++;

Trong ví dụ trên giá trị count được thay đổi 3 lần nhưng cùng 1 lần chạy chương trình và khi chương trình kết thúc, vậy sẽ có 1 lần thay đổi duy nhất:

VD: count = 0

=> count = 3

Tránh việc mỗi lần thay đổi là 1 lần render gây lãng phí tài nguyên

Trong lần chạy 1: Giá trị lấy ra từ DOM của count là 0, sau khi cập nhật thay đổi giá trị của count là 3 tuy nhiên trên DOM vẫn là 0 nếu chưa kết thúc chương trình thì sẽ không render lại, nếu bạn cố gắng truy cập count trong khi lần chạy đó chưa kết thúc, bạn chỉ nhận được giá trị 0 vì chưa thực hiện reder lại, do đó với textTick() bạn sẽ lấy được giá trị mới nhất của count nhưng không phải render luôn mà chờ kết thúc chương trình mới render lại. Tóm lại nextTick sẽ cho bạn lấy giá trị mới nhất trước khi nó được công bố

## computed

Khi giá trị tính toán chỉ tính khi giá trị phụ thuộc thay đổi, còn không thay đổi thì mỗi lần render lại sẽ lấy kết quả trước đó mà không cần tính toán lại, giống như ref, computed cũng cần phải được truy cập qua .value trong script. Computed giúp làm giảm khối lượng tính toán, và chỉ thực sự tính toán lại khi giá trị phụ thuộc thay đổi

Thuộc tính computed sẽ lưu vào bộ nhớ đệm dựa trên giá trị phụ thuộc

Thông thường hàm computed sẽ là chỉ đọc, tức là sau khi xử lý logic bên trong, ra bên ngoài computed bạn không thể thay đổi giá trị của nó. Tuy nhiên nếu bạn vẫn muốn ghi đè giá trị thì cần khai báo getter() và setter()

VD:

const totalPrice = computed({

get() {

return price.value \* quantity.value;

},

set(newValue) {

// Tính toán lại price dựa trên quantity, nếu không có newValue thì sẽ lấy giá trị quantity.value

price.value = newValue / quantity.value;

}

});

computed có thể cho chúng ta lấy giá trị trước đó là tham số đầu tiên của getter()

VD:

const totalPrice = computed((previousValue) => {

console.log('Previous Value:', previousValue);

return price.value \* quantity.value;

});

console.log(totalPrice.value); // Tính toán lần đầu, giá trị trước đó là undefined

price.value = 150;

console.log(totalPrice.value); // Tính toán lại, giá trị trước đó sẽ là 200

computed chỉ nên là side-effect dựa trên tính toán, không nên sử dụng cho thay đổi state khác, đồng bộ request, thay đổi DOM bên trong getter(), mà tất cả những điều này nên thực hiện một chức năng khác đó là watcher()

## Binding Class và style

Khi class và style là biến có giá trị động thay vì cố định, chúng ta sử dụng binding để làm điều này. Cú pháp là v-bind

VD:

<div :class="{ active: isActive }"></div>

Giải thích: nếu class có tên active mà là truthy thì khi render sẽ như sau:

<div class="active"></div>

Còn nếu là falsy thì sẽ:

<div></div>

Có thể nhận object

VD:

const classObject = reactive({

active: true,

'text-danger': false

})

Có thể kết hợp với computed() để tính toán giá trị

VD:

const isActive = ref(true)

const error = ref(null)

const classObject = computed(() => ({

active: isActive.value && !error.value,

'text-danger': error.value && error.value.type === 'fatal'

}))

Có thể dùng mảng

VD:

const activeClass = ref('active')

const errorClass = ref('text-danger')

<div :class="[activeClass, errorClass]"></div>

Có thể dùng toán tử 3 ngôi

VD:

<div :class="[isActive ? activeClass : '', errorClass]"></div>

Nếu định nghĩa component A và sử dụng trong component B, component A có class và component B gọi đến A lại thêm class nữa, thì lúc này sẽ render là gộp 2 class lại. Nhưng điều này chỉ áp dụng cho component A có 1 thẻ

VD:

MyComponent.vue

<!-- child component template -->

<p class="foo bar">Hi!</p>

// Khi sử dụng trong file khác

<!-- when using the component -->

<MyComponent class="baz boo" />

// Kết quả sẽ render ra:

<p class="foo bar baz boo">Hi!</p>

// tích hợp cả 2 class ở 2 bên

Nếu component A có nhiều thẻ hơn, thì bạn phải quy định thẻ nào thêm class bằng cách dùng biến $attrs

VD:

<template>

<div v-bind="$attrs">Content 1</div>

<p>Content 2</p>

</template>

khi render

<div v-bind="foo baz bar">Content 1</div>

<p>Content 2</p>

Tương tự với style cũng giống với class

VD: nếu font-size là '30'

<div :style="{ 'font-size': fontSize + 'px' }"></div>

hoặc (nếu font-size là '30px'

<div :style="{ 'font-size': fontSize + 'px' }"></div><div :class="{ active: isActive }"></div>

Giải thích: nếu class có tên active mà là truthy thì khi render sẽ như sau:

<div class="active"></div>

Còn nếu là falsy thì sẽ:

<div></div>

## Danh sách

### v-for

Sử dụng để lặp từng phần tử của mảng, sử dụng v-for cho thẻ nào thì thẻ đó sẽ lặp

Cú pháp:

<li v-for="item in items">

{{ item.message }}

</li>

// Trong đó mảng là items và item là bí danh (phần tử đại diện cho từng lần lặp)

v-for hỗ trợ thêm phần tử bí danh thứ 2 là index

VD:

<li v-for="(item, index) in items">

{{ parentMessage }} - {{ index }} - {{ item.message }}

</li>

Bạn cũng có thể phân giải cấu trúc (destructuring) biến bí danh nếu muốn dùng trực tiếp giá trị của biến đó (tuy nhiên không khuyến nghị vì sẽ hơi rối với mô hình lớn)

VD:

<li v-for="{ message } in items">

{{ message }}

</li>

<!-- with index alias -->

<li v-for="({ message }, index) in items">

{{ message }} {{ index }}

</li>

v-for hỗ trợ lồng nhau

v-for có thể lặp qua object bằng cách lặp qua từng key để lấy ra value(nếu chỉ 1 tham số) nếu đưa ra 2 tham số sẽ lấy lần lượt là (value, key), nếu 3 tham số sẽ là (value,key,index)

Nếu sử dụng v-for và v-if cùng 1 thẻ thì nó sẽ kiểm tra v-if trước nếu không thỏa mãn thì v-for sẽ không bao giờ xảy ra, thế nên việc kiểm tra điều kiện qua mỗi vòng lặp sẽ không được thực hiện, mà thay vào đó hãy cho v-if lồng bên trong v-for

lưu ý: nên sử dụng key để nó có thể xác định phần tử khác nhau giữa các phần tử, và key phải là phần tử duy nhất không thay đổi, nếu khi render lại mà key thay đổi thì dẫn đến lấy các phần tử không chính xác, và v-for đặt ở đâu thì key đặt tại vị trí đó

VD:

<div v-for="item in items" :key="item.id">

<!-- content -->

</div>

Có thể dùng v-for cho component tương tự như các thẻ khác, và điều này cũng không truyền dữ liệu cho đến component vì component được đóng gói riêng của nó, mình chỉ lặp và tái sử dụng nên không ảnh hưởng đến component

Có thể sử dụng v-bind trong quá trình lặp component

### Các phương pháp đột biến

Các phương pháp đột biến mảng bao gồm: push, pop, shift, unshift, splice, sort, reserse sẽ làm thay đổi mảng hiển thị. Tuy có một số phương thức không làm thay đổi mảng gốc nhưng khi hiển thị ra là một khác, và Vue sẽ không render lại toàn bộ danh sách mà chỉ render những phần tử nào thay đổi

### Lọc, sắp xếp

Đôi khi chúng ta chỉ cần lọc hoặc sắp xếp một mảng, ta nên dùng computed trước khi đưa vào lặp hiển thị.

Nếu là trường hợp kiểu tra trong mỗi lần lặp thì computed không khả thi, thay vào đó hãy tạo một hàm xử lý bên ngoài và mỗi lần lặp hãy gọi đến hàm xử lý

Lưu ý với reverse() và sort() sẽ làm thay đổi mảng gốc, do đó nên copy mảng gốc sang mảng mới rồi sử dụng phương thức này trên mảng mới

## Sử lý sự kiện

### Lắng nghe sự kiện

Dùng v-on để bắt sự kiện, có thể viết tắt là @

VD: v-on:click hoặc @click

Có 2 cách sử dụng là viết trực tiếp xử lý logic bên trong hoặc gọi đến hàm xử lý bên ngoài

Lưu ý: có thể dùng trực tiếp event của DOM như bình thường, tuy nhiên với 2 tham số trở lên thì không làm được như sau

VD:

const name = ref('Vue.js')

function greet(event) {

alert(Hello ${name.value}!)

// event is the native DOM event

if (event) {

alert(event.target.tagName)

}

}

<!-- greet is the name of the method defined above -->

<button @click="greet">Greet</button>

Với 2 tham số trở lên, lúc này event sẽ được gọi dưới dạng biến $event

VD:

<!-- using $event special variable -->

<button @click="warn('Form cannot be submitted yet.', $event)">

Submit

</button>

<!-- using inline arrow function -->

<button @click="(event) => warn('Form cannot be submitted yet.', event)">

Submit

</button>

### Tùy chỉnh sự kiện

Trong trường hợp bạn không muốn sự kiện diễn ra theo mặc định, bạn hoàn toàn có thể điều chỉnh nó

VD: Khi nhấn nút submit mặc định là load lại trang hoặc chuyển hướng đến trang khác, tuy nhiên với event.preventDefault() bạn có thể ngăn được điều này

Các trường hợp khác

+ event.stopPropagation()

<!-- the click event's propagation will be stopped -->

<a @click.stop="doThis"></a>

<!-- the submit event will no longer reload the page -->

<form @submit.prevent="onSubmit"></form>

<!-- modifiers can be chained -->

<a @click.stop.prevent="doThat"></a>

<!-- just the modifier -->

<form @submit.prevent></form>

<!-- only trigger handler if event.target is the element itself -->

<!-- i.e. not from a child element -->

<div @click.self="doThat">...</div>

<!-- use capture mode when adding the event listener -->

<!-- i.e. an event targeting an inner element is handled -->

<!-- here before being handled by that element -->

<div @click.capture="doThis">...</div>

<!-- the click event will be triggered at most once -->

<a @click.once="doThis"></a>

<!-- the scroll event's default behavior (scrolling) will happen -->

<!-- immediately, instead of waiting for onScroll to complete -->

<!-- in case it contains event.preventDefault() -->

<div @scroll.passive="onScroll">...</div>

Lưu ý không dùng .passive và .prevent cùng nhau

### Lắng nghe phím

Đôi khi bạn muốn bắt sự kiện cho các phím đầu vào từ bàn phím thay vì dùng chuột

VD: Enter, mũi tên trái, phải, ...

<!-- only call submit when the key is Enter -->

<input @keyup.enter="submit" />

<input @keyup.page-down="onPageDown" />

Sử dụng @keyup

Các nút thường dùng:

.enter

.tab

.delete

.esc

.space

.up

.down

.left

.right

Các nút điều khiển:

.ctrl

.alt

.shift

.meta

Có thể kết hợp các phím:

<!-- Ctrl + Click -->

<div @click.ctrl="doSomething">Do something</div>

Tuy nhiên nếu với ví dụ trên, khi nhấn Ctrl + bất kỳ phím nào khác cũng sẽ kích hoạt sự kiện Ctrl để ngăn chỉ có Ctrl được kích hoạt khi sử dụng 1 mình hãy sử dụng .exact

VD:

<button @click.ctrl.exact="onCtrlClick">A</button>

Các phím của chuột:

left

right

middle

## Form

Thông thường với js, bạn muốn kết nối trường nhập liệu của form với giá trị của biến và gán cho sự kiện thay đổi khi nhấn nút

VD:

<input :value="text" @input="event => text = event.target.value">

Điều này Vue có cách rút gọn đơn giản hơn

<input v-model="text">

Điều này thuận tiện cho việc liên tục cập nhật giá trị của đối tượng, và nó linh hoạt cho bất kỳ type input nào

v-model sẽ bỏ qua các giá trị khởi tạo ban đầu của các thuộc tính được thấy của phần tử mà nó sẽ ưu tiên lấy từ luồng sự kiện js

Tuy nhiên với một số dạng nhập ký tự đặc biệt IME (hàn quốc, trung quốc, nhật, ...) thì v-model không hỗ trợ tốt, nên dùng event thông thường của js

Không nên dùng v-model kèm nội suy cùng trong 1 thẻ

VD1:

<!-- bad -->

<textarea>{{ text }}</textarea>

<!-- good -->

<textarea v-model="text"></textarea>

VD2:

<input type="checkbox" id="checkbox" v-model="checked" />

<label for="checkbox">{{ checked }}</label>

Đối với select, mặc định chỉ chọn 1 trường, nhưng nếu thêm thuộc tính "multiple" sẽ có thể chọn nhiều hơn 1 khi giữ Ctrl + Click

VD:

<div>Selected: {{ selected }}</div>

<select v-model="selected" multiple>

<option>A</option>

<option>B</option>

<option>C</option>

</select>

Có thể kết hợp với v-for

VD:

const selected = ref('A')

const options = ref([

{ text: 'One', value: 'A' },

{ text: 'Two', value: 'B' },

{ text: 'Three', value: 'C' }

])

===

<select v-model="selected">

<option v-for="option in options" :value="option.value">

{{ option.text }}

</option>

</select>

<div>Selected: {{ selected }}</div>

Đối với checkbox sẽ có thêm 2 thuộc tính :true-value và :false-value nhận giá trị boolean tương ứng, hoặc cũng có thể là một chuỗi bạn quy định (:true-value="'yes'" :false-value="'no'")

### Modifiers

#### lazy

Bạn không muốn kiểm tra sau mỗi lần người dùng nhập liệu, bạn muốn sau khi người dùng nhập xong và di chuyển qua vùng khác mới kiểm tra thì .lazy là lựa chọn thích hợp

VD:

<!-- synced after "change" instead of "input" -->

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

#### number

Nếu bạn muốn tự động ép kiểu thành number bạn có thể thêm .number, nếu không ép được sẽ hiển thị dạng mặc định ban đầu

#### trim

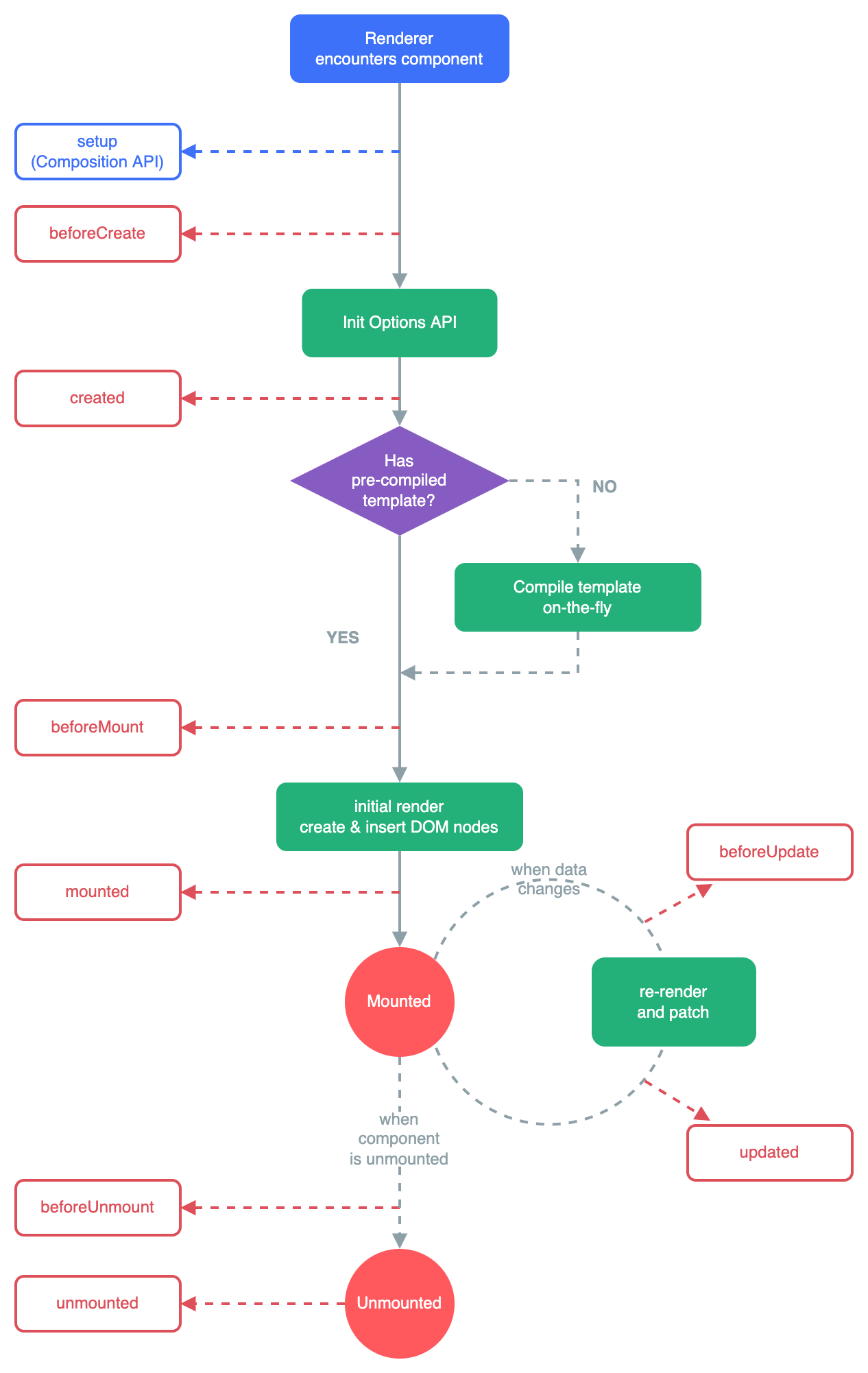
Loại bỏ các kí tự khoảng trắng (space, tab) ở đầu và cuối chuỗi

## Lifecycle Hooks

Trong 1 instance Vue sẽ có các giai đoạn khác nhau

Các giai đoạn bao gồm: Khởi tạo, mount, cập nhật, unmount

Có thể xem hình bên dưới để hiểu rõ hơn về mô hình



Bằng các hàm này, bạn có thể tùy chỉnh can thiệp vào quá trình thực hiện chương trình tại đúng các mốc thời điểm bạn muốn

Lưu ý: Khi hàm onMounted được gọi nó yêu cầu liên kết với hàm khởi tạo ngay lập tức, có nghĩa là nó phải được đồng bộ, nếu gọi trong hàm bất đồng bộ, nó sẽ không được thực thi

Xem thêm: <https://vuejs.org/api/composition-api-lifecycle.html>

## Watcher

### Watcher đơn giản

Là phần giống với computed tuy nhiên computed hướng tới việc tính toán là chính thì watcher sẽ thiên về xử lý hiển thị (side-effect) chẳng hạn như các thay đổi trên DOM, hoặc các thay đổi bất đồng bộ

VD:

<script setup>

import { ref, watch } from 'vue'

const question = ref('')

const answer = ref('Questions usually contain a question mark. ;-)')

const loading = ref(false)

// watch works directly on a ref

watch(question, async (newQuestion, oldQuestion) => {

if (newQuestion.includes('?')) {

loading.value = true

answer.value = 'Thinking...'

try {

const res = await fetch('https://yesno.wtf/api')

answer.value = (await res.json()).answer

} catch (error) {

answer.value = 'Error! Could not reach the API. ' + error

} finally {

loading.value = false

}

}

})

</script>

<template>

<p>

Ask a yes/no question:

<input v-model="question" :disabled="loading" />

</p>

<p>{{ answer }}</p>

</template>

Giải thích:

* Đoạn code trên sẽ xử lý gọi api kiểm tra đầu vào nếu người dùng nhập tới ký tự ? và đưa ra câu trả lời phù hợp
* Tham số thứ nhất của watcher là nguồn (ref hoặc reactive)
* Nếu muốn lấy 1 thuộc tính của reactive vào watcher bạn không thể truy cập trực tiếp thuộc tính đó, mà cần phải dùng getter của watcher

VD:

const obj = reactive({ count: 0 })

// this won't work because we are passing a number to watch()

watch(obj.count, (count) => {

console.log(Count is: ${count})

})

// instead, use a getter:

watch(

() => obj.count,

(count) => {

console.log(Count is: ${count})

}

)

### Watcher phức tạp

-Nếu thay vì chọn 1 thuộc tính của reactive mà chọn cả reactive thì watcher sẽ theo dõi toàn bộ từng phần tử có trong reactive kể cả lồng sâu, và điều này cũng tốn tài nguyên nên nếu thực sự cần thiết mới sử dụng

### Eager Watcher

Watcher sẽ không chạy nếu như nguồn của nó không thay đổi, tuy nhiên có thể cho nó chạy trước 1 lần trước khi kiểm tra sự thay đổi (giống với do while), bằng cách thêm tham số thứ 3 là { immediate : true }, điều này sẽ luôn đảm bảo chạy 1 lần khi khởi tạo

VD:

watch(

source,

(newValue, oldValue) => {

// executed immediately, then again when source changes

},

{ immediate: true }

)

### Once Watcher

Bằng cách thêm tham số { once : true} watcher sẽ chỉ chạy khi lần thay đổi đầu tiên của nguồn được thay đổi, còn những lần sau nó sẽ phớt lờ

### watcherEffect()

Theo dõi 2 đoan mã sau, có cùng chung 1 mục đích là lấy dữ liệu khi thay đổi id tương ứng

VD1:

const todoId = ref(1)

const data = ref(null)

watch(

todoId,

async () => {

const response = await fetch(

https://jsonplaceholder.typicode.com/todos/${todoId.value}

)

data.value = await response.json()

},

{ immediate: true }

)

VD2:

watchEffect(async () => {

const response = await fetch(

https://jsonplaceholder.typicode.com/todos/${todoId.value}

)

data.value = await response.json()

})

Có thể thấy đoạn 2 ngắn hơn, ở VD1 quy định nguồn toDoId làm biến phụ thuộc thì ở đoạn 2 sẽ theo

# VUE ROUTER

## Định nghĩa

Là bộ định tuyến cho vue phía máy khách

npm install vue-router@4

Chạy bởi các ứng dụng trang đơn SPA, điều hướng xung quanh ứng dụng không cần phải tải lại từ máy chủ

## Router cơ bản

2 thành phần chính của Vue Router bao gồm: RouterLink và RouterView

* RouterLink: giống thẻ a

<RouterLink to="" > </RouterLink>

* Nơi hiển thị thành phần định tuyến, thường được đặt ở vị trí cao nhất của dự án (App.vue, ...)

Sử dụng {{ $route.fullPath }} để hiển thị slug hiện tại:

VD:

<p>{{ $route.fullPath }}</p>

// result: /authorize

router.options.routes // hiển thị toàn bộ định tuyến

routes: {

path,

name

}

Ví dụ tạo 1 bộ định tuyến

import { createMemoryHistory, createRouter } from 'vue-router'

import HomeView from './HomeView.vue'

import AboutView from './AboutView.vue'

const routes = [

{ path: '/', component: HomeView },

{ path: '/about', component: AboutView },

]

const router = createRouter({

history: createMemoryHistory(),

routes,

})

Bộ định tuyến có phân nhánh nhỏ hơn (/authorize/signin)

import { createRouter, createWebHistory } from 'vue-router'

import PageDefault from '../Layouts/PageDefault.vue'

import AuthLayout from '../Layouts/AuthLayout.vue'

const router = createRouter({

history: createWebHistory(import.meta.env.BASE\_URL),

routes: [

{

path: '',

name: 'home',

component: PageDefault,

children: [

{

path: '',

name: 'homePage',

component: () => import('../views/Home.vue')

},

{

path: 'shop',

name: 'shop',

component: () => import('../views/Shop.vue')

},

{

path: 'about',

name: 'about',

component: () => import('../views/About.vue')

},

{

path: 'blog',

name: 'blog',

component: () => import('../views/Blog.vue')

},

{

path: 'contact',

name: 'contact',

component: () => import('../views/Contact.vue')

}

]

},

{

path: '/authorize',

component: AuthLayout,

children: [

{

path: 'signin',

name: 'signin',

component: () => import('../views/authorize/SignIn.vue')

},

{

path: 'signup',

name: 'signup',

component: () => import('../views/authorize/SignUp.vue')

}

]

}

]

})

export default router

Đăng ký bộ định tuyến

Sau khi tạo phiên bản bộ định tuyến, cần tạo plugin bằng cách dùng use() trên ứng dụng

App.ts hoặc App.js

import { createApp } from 'vue'

import App from './App.vue'

import router from './router'

const app = createApp(App)

app.use(router)

app.mount('#app')

Kết luận: Các bước xây dựng bộ định tuyến:

* B1: Đăng ký toàn cục: RouterView và RouterLink (App.vue)
* B2: Thêm các thuộc tính cho router và routes (thêm các page)
* B3: Khai báo và sử dụng router tại ứng dụng (App.ts or App.js)

## Router với tham số

## Routes khớp so với cú pháp (Match Regex)

## Đặt tên Routes

Có thể sử dụng name thay vì path trong thuộc tính to của RouterLink

## Routes lồng nhau

Sử dụng thuộc tính children

const routes = [

{

path: '/user/:id',

component: User,

children: [

{

// UserProfile will be rendered inside User's <router-view>

// when /user/:id/profile is matched

path: 'profile',

component: UserProfile,

},

{

// UserPosts will be rendered inside User's <router-view>

// when /user/:id/posts is matched

path: 'posts',

component: UserPosts,

},

],

},

]

## Redirect and Alias

Redirect: Điều hướng sang url khác được chỉ định

Alias: Tất cả url đều dùng chung 1 component (khác url chung nội dung)

# MẸO

## Prettier

.prettierrc.json

{

"$schema": "https://json.schemastore.org/prettierrc",

"semi": true,

"singleQuote": true,

"printWidth": 200

}

## Cấu hình file .env

Nơi chứa các cấu hình sử dụng toàn bộ dự án

.env: cấu hình chung

.env.development: cấu hình cho dev, thường chạy phía local, sử dụng khi chạy lệnh npm run dev

.env.production: cấu hình cho sản phẩm cuối, thường chạy bên server chính, sử dụng khi chạy lệnh npm run server, npm run build

Sử dụng .gitignore để chọn những file nào sẽ đẩy lên git file nào không

VD:

# file .env

VITE\_API\_BASE\_URL="https://api.abc.com"

VITE\_AUTHORIZE\_NAME="Authorization"

VITE\_APP\_TITLE="Identity"

VITE\_GOOGLE\_SITE\_KEY="6LfAOfQnAAAAADSoRyyWIzoaUeMDLvyDft0q0nG7s"

# file .env.development

VITE\_APP\_MODE="development"

# file .env.production

VITE\_APP\_MODE="production"

Cách sử dụng:

import.meta.env.VITE\_API\_BASE\_URL

// kết quả: https://api.abc.com

Tùy thuộc vào môi trường, nó sẽ quyết định lấy giá trị của biến thuộc file nào

## Cấu hình axios

Sử dụng cho việc gọi api

Bước 1: Tạo axios custom từ nguồn axios gốc

axios.defaults.baseURL = import.meta.env.VITE\_API\_BASE\_URL;

export const axiosDefault = axios.create()

Như vậy chúng ta đã có một hàm axiosDefault y chang axios gốc

Trong hầu hết các api khi gửi yêu cầu sẽ cần mã xác thực token để gửi kèm request, và bạn có thể sử dụng interceptors của axios để đính kèm token cho tất cả các request

Bước 2: Cấu hình url axios mặc định

axiosDefault.defaults.baseURL = import.meta.env.VITE\_API\_BASE\_URL;

Như vậy mỗi request bạn gửi đi không cần khai báo địa chỉ gốc của url mà chỉ cần thêm slug

Bước 3: Cấu hình axios request

VD:

axiosDefault.interceptors.request.use(

(config: InternalAxiosRequestConfig) => {

const token = getAuthorization();

if(token) {

config.headers['Authorization'] = `Bearer ${token}`;

}

return config;

},

(error) => {

return Promise.reject(error);

}

)

request sẽ nhận 2 tham số, một tham số trả về config nếu đúng còn sai thì sẽ trả về error

Bước 4: Cấu hình axios response

VD:

axiosDefault.interceptors.response.use(

(response: AxiosResponse) => {

return response;

},

(error: any) => {

// if error 401 and not include auth api return login

if(error.response.status === 401 && !whiteList.includes(error.config.url)) {

{

delLocalStorageUserData();

window.location.href='/login';

}

}

}

)