好用套件

- 選擇時間篩選工具

    https://vue3datepicker.com/

    ![Untitled.png](https://prod-files-secure.s3.us-west-2.amazonaws.com/87b495d8-0bc3-49b6-9777-8698fe91b622/47556d9f-3f51-4965-8240-1e805401ff7e/Untitled.png?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Content-Sha256=UNSIGNED-PAYLOAD&X-Amz-Credential=ASIAZI2LB466SXDHNLIS%2F20251204%2Fus-west-2%2Fs3%2Faws4_request&X-Amz-Date=20251204T164616Z&X-Amz-Expires=3600&X-Amz-Security-Token=IQoJb3JpZ2luX2VjEID%2F%2F%2F%2F%2F%2F%2F%2F%2F%2FwEaCXVzLXdlc3QtMiJHMEUCICqBVwGsN1PQBwAAOjijN%2BUkISC2aiWhSgEWoOJcxrg1AiEAtBOnxs6GGJR5JT7n5a%Amz-Signature=6e3e8bf56b07fa9f8866ebc635f1421540be31171f918617cb39116e1c9a6379&X-Amz-SignedHeaders=host&x-amz-checksum-mode=ENABLED&x-id=GetObject)

- 輸入文字會出現 Tag 效果

    https://vuejsexamples.com/a-smart-input-tags-component-with-vue-3/

    ![Untitled.png](https://prod-files-secure.s3.us-west-2.amazonaws.com/87b495d8-0bc3-49b6-9777-8698fe91b622/987f7718-7e0d-4d97-83f3-f1b87169738f/Untitled.png?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Content-Sha256=UNSIGNED-PAYLOAD&X-Amz-Credential=ASIAZI2LB4663QRJYWTF%2F20251204%2Fus-west-2%2Fs3%2Faws4_request&X-Amz-Date=20251204T164616Z&X-Amz-Expires=3600&X-Amz-Security-Token=IQoJb3JpZ2luX2VjEID%2F%2F%2F%2F%2F%2F%2F%2F%2F%2FwEaCXVzLXdlc3QtMiJIMEYCIQD1bD4WqykzmZYtVovbfovpO7Lu9Ivi33cjUg%2BdrW2yiwIhAOzYACzVSGy7NymFzZRAmz-Signature=b8667090f7c84ae60fc70fd498e6bb242bd814dd6f8eb0b1caa3ca72e34b1416&X-Amz-SignedHeaders=host&x-amz-checksum-mode=ENABLED&x-id=GetObject)

- 圖表工具 Vue3 + Chart.js

    [//]: # (child_page is not supported)

- NVM 切換不同版本的 node.js

    因為有些 Vue 專案使用的會是不同版本的 node.js 所以會需要自由切換 node version 所以使用 NVM node 版本管理工具

    `!!!注意!!!` 如果不是使用 「nvm install {node version}」的方式安裝，在切換的時候可能會無法順利切換成功，因此建議使用 NVM 的方式安裝版本。此外如果還是有切換不順的問題，可以把所有 node、nvm 都重新刪掉，然後重新安裝一下 nvm 的流程。

    操作步驟:

    1. 先安裝 NVM

可以直接使用 Windows exe 進行安裝

    https://github.com/coreybutler/nvm-windows/releases

        ![image.png](https://prod-files-secure.s3.us-west-2.amazonaws.com/87b495d8-0bc3-49b6-9777-8698fe91b622/7125f193-a27d-442f-8b79-77078a00fdc1/image.png?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Content-Sha256=UNSIGNED-PAYLOAD&X-Amz-Credential=ASIAZI2LB466TYW67OME%2F20251204%2Fus-west-2%2Fs3%2Faws4_request&X-Amz-Date=20251204T164617Z&X-Amz-Expires=3600&X-Amz-Security-Token=IQoJb3JpZ2luX2VjEID%2F%2F%2F%2F%2F%2F%2F%2F%2F%2FwEaCXVzLXdlc3QtMiJHMEUCIDb%2FIJe0WtCmRJHDnj3SStfHSFM7nJK1NU85sug4ivMCAiEA%2Bu1s%2FhsxkjlIFCVAmz-Signature=a7871f1bdfcf1c1f1ec009b25d986375ee7951014f1d0d40892e917442b53a58&X-Amz-SignedHeaders=host&x-amz-checksum-mode=ENABLED&x-id=GetObject)

    2. 在 terminal 中輸入「nvm --version」確認安裝是否成功

    3. 下載想要的 node.js 版本，並在 termainal 中輸入「nvm ls」 查看目前所有的 node.js 版本

    4. 在 Terminal 中輸入「nvm use {your node version}」如「nvm use 14.16」，就可以輸入「npm --version」查看是否成功切換版本

- 增加 Chrome的 Vue Debug 套件

    安裝 Vue.js devtools https://devtools.vuejs.org/guide/installation.html

    ![Untitled.png](https://prod-files-secure.s3.us-west-2.amazonaws.com/87b495d8-0bc3-49b6-9777-8698fe91b622/c8ff8a27-370a-4445-bfd0-804c84834a7e/Untitled.png?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Content-Sha256=UNSIGNED-PAYLOAD&X-Amz-Credential=ASIAZI2LB46647XC7UBW%2F20251204%2Fus-west-2%2Fs3%2Faws4_request&X-Amz-Date=20251204T164617Z&X-Amz-Expires=3600&X-Amz-Security-Token=IQoJb3JpZ2luX2VjEID%2F%2F%2F%2F%2F%2F%2F%2F%2F%2FwEaCXVzLXdlc3QtMiJIMEYCIQD9uQFthCcB6WwL%2BhS3fKBYaS%2BFQ0oIuo9UWXkAv29nDwIhANOBF2ZZn5b0283GeAmz-Signature=db0367e217aba5e205b451cee9f860dc70ce2435cbd84bfeaa7f858b1a5f05d3&X-Amz-SignedHeaders=host&x-amz-checksum-mode=ENABLED&x-id=GetObject)

排程規劃

- node-cron 排程 Call API 基礎應用(TS)

    - 設定

        在專案中安裝套件

        1. 建立專案，如果已經建立可以不需要這個步驟「npm install typescript ts-node」

        1. 安裝套件 「npm install node-cron」

        1. 設定使用文件「npm install @types/axios @types/node-cron」

        - `**@types/axios**` 是 axios 模組的型別定義，用於告訴 TypeScript 如何處理 axios 模組的型別。

        - `**@types/node-cron**` 是 node-cron 模組的型別定義，用於告訴 TypeScript 如何處理 node-cron 模組的型別。

    - 腳本撰寫

        1. 在 root 設定 tsconfig.json 檔案，內容如下

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs",
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true
  }
}
```

        1. 建立一個 “src” folder，創建一個 schedule.ts 腳本，內容中放入程式碼

```
import axios from 'axios';
import * as cron from 'node-cron';
//* as cron: 這表示將整個模組的內容引入一個名為 cron 的物件中。這樣，你就可以透過 cron 這個物件來訪問 node-cron 模組中的所有功能。

const executeAPI = async () => {
  try {
        // 這邊是放入要執行排程的 API
    const response = await axios.get('http://localhost:4000/model-request');

    console.log('API 執行成功', response.data);
  } catch (error) {
    console.error('API 執行失敗', error);
  }
};

cron.schedule('* * * * *', () => {
  console.log('排程執行+1');
  executeAPI();
});

console.log('排程已啟動');
```

    - 啟用腳本

        1. 啟用腳本，在 Termainl 中輸入「npx ts-node src/schedule.ts」啟用排程

- 更符合使用情境的排程規劃(JS)

    【需求 / 實作】

    1. 每天拿新的資料儲存進資料庫

        1. 創建新的 collection 做到儲存進資料庫，以取資料後比對

    1. 方便的新增需求與規則的方法

        1. 在 config 檔(?)中，建立一個規則，讓排程程式在執行前先跑過一遍確認規則，以此確保規則更新

    1. 排程日誌紀錄

        1. 單純使用 .log 文件記錄

    1. 排程錯誤發生的時候，會需要通知

        1. 寄送出 email 或是串接到 Slask / Line 上通知

- 查詢研究

    celery python cron

    https://medium.com/the-andela-way/crontabs-in-celery-d779a8eb4cf

    https://w3c.hexschool.com/blog/d6668f69

    比較大型的，由 Airbnb 推出開源系統 https://airflow.apache.org/

TypeScript

- as const 型別斷言，將元素變得不能被修改

`**as const**` 是一個 TypeScript 的型別斷言（Type Assertion）的語法，會將陣列中的每個元素內的值，設定成只能是當前的值，不能被重新賦值或修改。可以幫助你在代碼中更嚴格地使用常數，並防止意外的修改

```
範例一、
const daysOfWeek = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday'] as const;
==============
範例二、
const direction = 'left' as const; // direction 的型別是 'left'
==============
範例三、
const config = {
  API_KEY: 'your-api-key',
  MAX_RESULTS: 10,
} as const;Node.js
```

- 型別設定，因為是靜態語言所以需要將每個變數都固定起來

as string

XXX :string

const modelDemandsData = ref<Array<object>>([]);

- 型別設定

文章出處:[https://www.typescriptlang.org/zh/docs/handbook/typescript-in-5-minutes.html]
(https://www.typescriptlang.org/zh/docs/handbook/typescript-in-5-minutes.html)

- 單一字串設定

```
let helloWorld: string = "Hello World”
```

- 物件型別設定

```
interface User {
  name: string;
  id: number;
}

const user: User = {
  name: "Hayes",
  id: 0,
};
```

- Function 設定

```
interface User {
  name: string;
  id: number;
}

function getAdminUser(): User {
  //...
}

function deleteUser(user: User) {
  // ...
}
```

- 組合類型設定(聯合 & 泛型)

- 聯合

如果型別使用聯合類型的型別，就可以限制變數只能使用特定的值，這樣可以有效的確保了變數的合法值，並且幫助捕獲潛在的代碼錯誤。

```
type MyBool = true | false;
type WindowStates = "open" | "closed" | "minimized";
type LockStates = "locked" | "unlocked";
type PositiveOddNumbersUnderTen = 1 | 3 | 5 | 7 | 9;
```

聯合也提供一種處理多個型別的方法

```
// 這個 function 可以同時處理 array & string
function getLength(obj: string | string[]) {
  return obj.length;
}
```

- 泛型

泛型允許你在定義程式碼時引入可變的型別，從而提高程式碼的重用性和靈活性。這對於創建通用、可適應不同情況的程式碼非常有用。泛型可以應用在函數、類別、介面等不同地方，以提供更通用和靈活的型別支援。

```
type StringArray = Array<string>;
type NumberArray = Array<number>;
type ObjectWithNameArray = Array<{ name: string }>;
```

- 以函數使用為例

```
function identity<T>(arg: T): T {
  return arg;
}

// 使用方式
let result = identity<string>("Hello, TypeScript");
====================
console.log(result)  => Hello, TypeScript

// 備註:
// 這邊的 T 是一個型別參數，它代表了一個未知的型別，
// 所以下面正式使用的時候，我們用 string 取代，如果沒有寫的話，系統會跳緊告
// 相同的概念，通常會使用 T、U ... 但意義都是代號而已，
// 所以通常會更具描述性的名稱，例如 Type 或 Value
```

- 以泛型類別為例

```
class Box<T> {
  private value: T;
  constructor(value: T) {
    this.value = value;
  }
  getValue(): T {
    return this.value;
  }
}

// 使用方式
let numberBox = new Box<number>(42);
let stringBox = new Box<string>("Hello, Generics");
```

- 以泛型介面為例

```
interface Pair<T, U> {
  first: T;
  second: U;
}

// 使用方式
let pair: Pair<number, string> = { first: 1, second: "two" };
```

- 以泛型約束為例

```
// logLength 函數使用了一個泛型型別 T
// 這個泛型型別受到約束。約束的內容是 { length: number }
// T 必須是一個物件型別，並且這個物件型別必須包含一個名為 length 的屬性，
// length 這個屬性的值必須是數字。

function logLength<T extends { length: number }>(arg: T): void {
  console.log(arg.length);
}

// length 屬性的意思就是需要可以數的意思，例如 String、Array

// 使用方式
logLength("Hello, Generics"); // 合法
logLength([1, 2, 3]); // 合法，因为数组也有 length 属性
logLength(42); // 錯誤，因為數字沒有 length 屬性
```

- 結構化的類型系統 TOBE

Vue 基礎操作

互動教學連結:https://cn.vuejs.org/tutorial/#step-1

- fs 模塊(模組) 用來操作文件模塊，滿足文件操作的需求

    fs.readFile() >> 讀取指定文件中的內容

    fs.writeFile() >>將指定文件中寫入內容

    調用方法:const fs = require(”fs”) >> 解釋:調用 require 方法，給一個 fs 字串，用來明確說明我們要導入 fs 模組，接下來使用 const 常量 接受 fs，之後就可以調用 fs 模組內的 readFile() 或是 writeFile()，

    require 與 python 的 既有 function 很像，例如 .mean()

    const 與 python 的變數很像，但除了可以接 function 外，還可以接一個計算結果，例如:const fs = require(”fs”) 或是 const arrOld = dataStr.split(” ”)，一個是接住 function 然後可以拿來使用功能，一個是接住一個結果


```
fs.readFile(path, [option], callback)

const fs = require("fs")
fs.readFile("./files/11.txt", "utf-8", function(err, dataStr){
    console.log(err)
    console.log("------------")
    console.log(dataStr)
})

// 如果有讀取成功，會顯示 dataStr 的結果，如果失敗，會呈現出 err 結果

fs.writeFile(path, data[...,option], callback)
```

- Function 與 Python Function 比較

    | Python | Node |

| ---- | ---- | | item_list = [ ] for item in (List / Array): item = item.replace(”=”, “: ”) item_list . append(item) print(item_list) | const arrNew = [ ] arrOld . forEach (item => { arrNew.push(item.replace("=", ": ")) }) console.log(arrNew) | | function resolveCSS( ) { } | def resolveCSS( ): | | const A = XXXX | A = XXX |

- 箭頭符號「= >」

    是一個縮減程式碼的新功能

    `const func = (x) => x + 1`

    展開如下，等於是 function 與 return 的縮減

    `const func = function (x) { return x + 1 }`

    注意，有無大括號有差異:

```
const funcA = x => x + 1
const funcB = x => { x + 1 }

//==============================
funcA(1) //2
funcB(1) //undefined
```

- `reactive()` API 来声明响应式状态

    import { reactive } from 'vue'   `reactive()` API 来声明响应式状态

```
const counter = reactive({
  count: 0
})

console.log(counter.count) // 0
counter.count++
console.log(counter.count) // 1
```

- `ref()` 則可以接受任何值类型

    import { ref } *from* 'vue'

```
const message = ref('Hello World!')

console.log(message.value) // "Hello World!"
message.value = 'Changed'
console.log(message.value) // "Changed"
```

在程式設計中，`**ref**`（參考）通常指的是一個指向某個變數或物件的引用，而不是直接存儲其值。這個概念在不同的程式語言和上下文中可能有不同的用途，以下是一些常見的情況:

1. **傳遞參數或值給函式:** 當你想要將某個變數或物件傳遞給函式，並且希望在函式內部修改該變數或物件的值，你可以使用參考（`**ref**`）來實現。這樣做可以讓函式直接修改原始變數或物件，而不是創建一個新的拷貝。

1. **修改外部範疇的變數:** 在某些程式語言中，你可以使用參考（`**ref**`）來修改外部範疇（例如函式外部）的變數。這可以在一些情況下避免使用全域變數或傳遞大量參數。

1. **修改物件的屬性:** 當你傳遞一個物件給函式，而該函式需要修改物件的某些屬性時，你可以使用參考（`**ref**`）來實現這種修改，而不需要返回修改後的物件。

1. **實現指標類似的行為:** 在某些程式語言中，特別是低階語言，參考（`**ref**`）可以用來實現類似指標的行為，允許你直接訪問記憶體位置，從而進行更靈活的操作。

需要注意的是，`**ref**` 的用法和行為在不同的程式語言中可能有所不同。例如，在C++中，你可以使用指標來實現類似的功能；在C#中，`**ref**` 關鍵字可以用來傳遞參數的引用；在JavaScript中，則沒有直接的 `**ref**` 關鍵字，但你可以通過物件引用來達到類似的效果。因此，要根據你使用的程式語言和具體情況來理解和應用 `**ref**` 的概念。

- **Attribute **`v-bind`** 绑定一个动态值如(id、class、msg(自定義)…)，並擁有效果**

    透過下面的 style CSS ，透過 ref 定義後，以 class 的方式在 h1 中使用

【Class】
```
<script setup>
import { ref } from 'vue'

const titleClass = ref('title')
</script>

<template>
  <h1 :class="titleClass">Make me red</h1>
</template>

<style>
.title {
  color: red;
}
</style>
```

【id】
```
<div :id="dynamicId"></div>
```
【msg】
```
const greeting = ref('Hello from parent')
<ChildComp :msg="greeting" />
```
結果:
```
Hello from parent
// ref 讓您可以在 Vue 程式中創建一個具有響應性的變數，
// 使得數據的變動能夠在模板中得到及時反映。

<span style='color:orange'>如果沒有使用 ref 就不法使用 {{greeting}} 等帶參數的方式執行程式

冒號，動態帶變數</span>
```

`**id**` 與 `**class**` 是在 HTML 語言中用於標示元素的屬性，它們有以下的差異:

唯一性: `**id**` 屬性用於指定元素的唯一識別符。在整個 HTML 文件中，每個元素的 `**id**` 應該是唯一的，不能重複使用相同的 `**id**`。這使得 `**id**` 非常適合用於識別特定的元素，例如在 JavaScript 中進行 DOM 操作或跳轉到特定錨點。

群組性: `**class**` 屬性用於將多個元素進行分組。多個元素可以共用相同的 `**class**`，也可以同時擁有不同的 `**class**`。這使得 `**class**` 非常適合用於樣式化（CSS）和 JavaScript 操作一組元素。

- **事件监听 **`v-on` > @ **(** 指令监听 DOM**)**

```
<script setup>
import { ref } from 'vue'

const count = ref(0)
function increment(){
  count.value++
}
</script>
```

```
<template>
  <button @click="increment">count is: {{ count }}</button>
</template>
```

- DOM (Document Object Model) 文件物件模型

把一份 **HTML** 文件內的各個標籤，包括文字、圖片等等都定義成物件，將這些物件表示成結構化的物件樹，與爬蟲過程中，需要階層式的查詢資訊一樣，會需要用到 class、id…等等 elements

DOM 的主要功能包括:

1. **元素表示:** DOM 將 HTML 或 XML 文件中的每個元素（例如標籤、屬性、文本等）都表示成物件，使得程式可以透過這些物件來訪問和操作元素的屬性和內容。

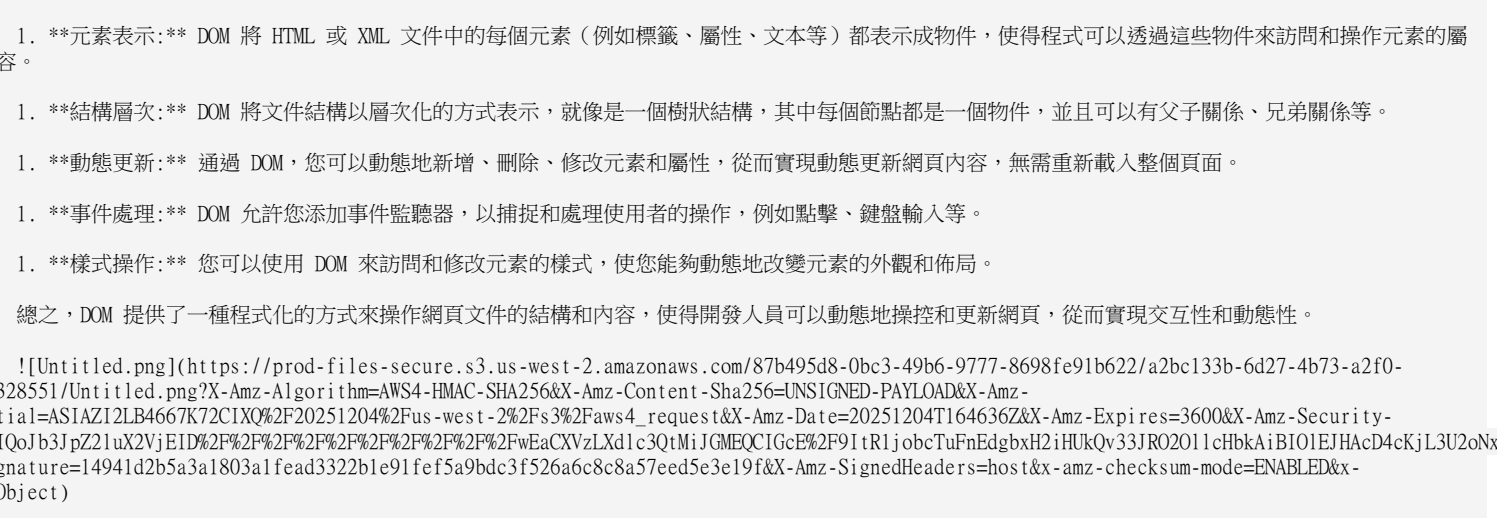
1. **結構層次:** DOM 將文件結構以層次化的方式表示，就像是一個樹狀結構，其中每個節點都是一個物件，並且可以有父子關係、兄弟關係等。

1. **動態更新:** 通過 DOM，您可以動態地新增、刪除、修改元素和屬性，從而實現動態更新網頁內容，無需重新載入整個頁面。

1. **事件處理:** DOM 允許您添加事件監聽器，以捕捉和處理使用者的操作，例如點擊、鍵盤輸入等。

1. **樣式操作:** 您可以使用 DOM 來訪問和修改元素的樣式，使您能夠動態地改變元素的外觀和佈局。

總之，DOM 提供了一種程式化的方式來操作網頁文件的結構和內容，使得開發人員可以動態地操控和更新網頁，從而實現交互性和動態性。



- **表单绑定 **`v-model` 簡化設定輸入以及呈現輸入內容

```
<script setup>
import { ref } from 'vue'

const text = ref('')

function onInput(e) {
  text.value = e.target.value
}
</script>

<template>
  <input :value="text" @input="onInput" placeholder="Type here">
  <p>{{ text }}</p>
</template>

== 簡化成以下程式碼 ==
<script setup>
import { ref } from 'vue'

const text = ref('')
</script>

<template>
  <input v-model="text" placeholder="Type here">
  <p>{{ text }}</p>
</template>
```

- **条件渲染 **`v-if`、`v-eles` 進行布林值切換

```
<script setup>
import { ref } from 'vue'

const awesome = ref(true)

function toggle() {
  awesome.value = !awesome.value
}
// 先定義 awesome=true，然後下面 toggle 按鈕如果被激發，則 awesome 變成不是 true=false
// 然後下面使用 v- if、else 的方式進行切換

</script>

<template>
  <button @click="toggle">toggle</button>
  <h1 v-if="awesome">Vue is awesome!</h1>
  <h1 v-else>Oh no 😢</h1>
</template>
```

- **列表渲染 **`v-for`** 跌代 + **push( ) and filter( )

【跌代寫法】

```
<ul>
<li v-for="todo in todos" :key="[todo.id](http://todo.id/)">
{{ todo.text }}
</li>
</ul>
```

等於是跌代 todos 然後將每一個 todo 設定成為一的 id 編碼，然後 使用 text type 的形式呈現出來

【filter( )】

這個箭頭函式通常用於 `**filter**` 方法或其他需要一個回傳值為布林型的函式的場景中。在 `**filter**` 方法中，當這個箭頭函式回傳值為 `**true**` 時，表示該元素會被保留在陣列中；而當回傳值為 `**false**` 時，表示該元素會被從陣列中過濾掉。

總結來說，`**(t) => t !== todo**` 是一個使用參數 `**t**` 來進行判斷操作的箭頭函式，通常用於陣列的 `**filter**` 方法來過濾陣列中的元素。

```plain text function removeTodo(todo) { todos.value = todos.value.filter((t) => t !== todo) }
```

【push( )】

```plain text
function addTodo() {
  todos.value.push({ id: id++, text: newTodo.value })
  newTodo.value = ''
}
```

在原本就存在的 todos list 中推入一些 valus 格視為 {id: XXXX, text:XXXX}

推入後，將 newTodo 的輸入空間清空

【完成程式碼】

```
<script setup>
import { ref } from 'vue'

// 给每个 todo 对象一个唯一的 id
let id = 0

const newTodo = ref('')
const todos = ref([
  { id: id++, text: 'Learn HTML' },
  { id: id++, text: 'Learn JavaScript' },
  { id: id++, text: 'Learn Vue' }
])

function addTodo() {
  todos.value.push({ id: id++, text: newTodo.value })
  newTodo.value = ''
}

function removeTodo(todo) {
  todos.value = todos.value.filter((t) => t !== todo)
}
</script>

<template>
  <form @submit.prevent="addTodo">
    <input v-model="newTodo">
    <button>Add Todo</button>
  </form>
  <ul>
    <li v-for="todo in todos" :key="todo.id">
      {{ todo.text }}
      <button @click="removeTodo(todo)">X</button>
    </li>
  </ul>
</template>
```

- **计算属性**

- **生命周期和模板引用 OnMounted async await**

- 透過程式碼理解

```
// 這一段程式碼就是使用異步操作的方式，拿到資料
// 其實我們平常使用@click=特定的 function 其實就是生命週期，只是下面操作的方法，
// 只是用來獲取資料而已，所以會在程式啟動的時候就執行，而不用在特定情況下(click)
// 被激活使用。
onMounted(async () => {
    const modelList = await getModelList();
    filterTagList.value = await getModelFilterList();
    // 備註，通常 API 會需要使用 await ，
    // 也因為為了提升整體 function 的效率所以使用 async
    modelList.map((model) => {
        models.value.push({
            coverUrl: model.coverUrl,
            modelName: model.modelName,
            summary: model.summary,
            credit: model.credit,
            unit: model.unit,
            description: model.description,
            modelID: model.modelID,
            tags: model.tags,
            author: model.author,
        });
    });
    modelsOrigin.value = models.value;
    filterTag.value.setFilterTag(filterTagList.value);
});
```


參考文章:[vue3 Composition API 學習手冊-13 生命週期 - iT 邦幫忙::一起幫忙解決難題，拯救 IT 人的一天 (ithome.com.tw)](https://ithelp.ithome.com.tw/articles/10242633)

![Untitled.png](https://prod-files-secure.s3.us-west-2.amazonaws.com/87b495d8-0bc3-49b6-9777-8698fe91b622/52460707-e879-46dd-8f95-73039b585f2f/Untitled.png?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Content-Sha256=UNSIGNED-PAYLOAD&X-Amz-Credential=ASIAZI2LB4663BQAWW3M%2F20251204%2Fus-west-2%2Fs3%2Faws4_request&X-Amz-Date=20251204T164638Z&X-Amz-Expires=3600&X-Amz-Security-Token=IQoJb3JpZ2luX2VjEID%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F%2FwEaCXVzLXdlc3QtMiJHMEUCIDvwsO0iZvVkmv7%2B2WuiII5KIATnLRjU%2F1dsRUkX%2FwzdAiEAxVAEWVb5znMM2xzAmz-Signature=1f73e44234aae82b0bb59f0964caeeee89c2d5bbc2d570ff2995323fe98cda6e&X-Amz-SignedHeaders=host&x-amz-checksum-mode=ENABLED&x-id=GetObject)

```
*import* { onMounted } *from* 'vue'

onMounted(() => {

*// 此时组件已经挂载。*

})
```

這個功能的用途是在組件初始化時執行一些異步操作或初始化邏輯，通常是需要在組件渲染後進行的事情。

當你需要在組件被插入到 DOM 中之後，進行一些資料的加載、初始化，或者與其他外部資源進行交互時，你可以使用 `**onMounted**`

- * DOM(Document Object Model)

網頁文件解析成一個由節點和物件組成的樹狀結構，並且每個節點代表了文檔中的一個部分，可以理解為一個表示網頁結構的抽象模型，它允許開發者使用程式語言（如 JavaScript）

DOM 是瀏覽器提供的一種 API，它使開發者能夠以編程方式操作和交互網頁的結構和內容。通過操作 DOM，你可以實現豐富的互動性和動態效果，並將網頁變得更加動態和有趣。

通過 DOM 可以做到以下:

1. **訪問元素:** 你可以通過 DOM 操作來獲取、查找、修改和刪除網頁中的元素，例如改變元素的內容、樣式、屬性等。

1. **創建元素:** 你可以使用 DOM 方法創建新的 HTML 元素，然後將它們添加到網頁中。

1. **處理事件:** 你可以註冊事件監聽器，以便在特定事件發生時執行程式碼，例如點擊、鍵盤輸入等。

1. **動態更新:** 你可以通過 DOM 在不重新加載整個頁面的情況下更新部分頁面內容，實現動態效果和交互。

1. **操作樣式:** 你可以使用 DOM 改變元素的樣式，包括位置、大小、顏色等。

- *異步操作 `**async/await**` 和 `**.then()**` `**.catch()**`

在程式開發中，異步處理是一種重要的概念，特別是當您需要從外部源（例如API、資料庫、網絡請求等）獲取資料時。以下是幾個原因解釋為什麼在拿取API資料時需要進行異步處理:

1. **非同步操作**:當您向API發送請求時，伺服器需要時間來處理請求並返回資料。如果在主線程中同步執行此操作，應用程序將被阻塞，直到請求完成為止，這可能會導致UI凍結，使應用無法對用戶進行交互。

1. **效能**:使用異步處理可以提高應用程序的效能，因為它允許其他操作在等待請求完成時繼續執行。這在處理大量數據或多個請求時特別重要。

1. **網絡延遲**:網絡請求的時間可能會受到網絡延遲等因素的影響。使用異步處理可以確保應用程序能夠正確處理請求的回應，而不需要等待整個請求過

程完成。

1. **用戶體驗**:異步處理有助於提供更好的用戶體驗。例如，您可以在請求的同時顯示一個載入指示器，以向用戶顯示正在進行操作，而不必等待請求完成。

您提供的程式碼是使用了 `**async/await**` 和 `**.then()**` 進行的異步處理範例。這種方式使得在請求完成之前，代碼可以繼續執行其他操作，同時也可以處理請求的回應。您的程式碼使用了Axios庫來發送GET請求並處理回應。其中的 `**.catch()**` 部分用於處理請求失敗的情況，例如網絡問題。

總之，異步處理在獲取外部資料時是一個重要的概念，它能確保您的應用程序在處理請求的同時保持活躍，提供更好的用戶體驗，並避免阻塞主線程。

- 程式範例

```
const getRequestDetail = async (modelRequestID) => {
    const output = await axios
        .get(DX_API.REQUEST_DETAIL.replace(':modelRequestID', modelRequestID))
        .then((response) => {
            if (response.status !== StatusCodes.OK) {
                throw new Error(response.data);
            }
            return response.data;
        })
        .then((data) => {
            return data.data;
        })
        .catch((error) => {
            console.log(error);

            return {};
        });

    return output;
};
```

- **偵听器 watch() 用來間空數據變化，然後進行回傳數值的調****整**

比較有問題的是，為什麼變動的只有 id 為甚麼還需要加入 fetchData？多加了解 fetchData 的使用，ChildComp

- **子組件 Props**** ****Emits ****【練習】**

![Untitled.png](https://prod-files-secure.s3.us-west-2.amazonaws.com/87b495d8-0bc3-49b6-9777-8698fe91b622/46d99a5f-33a4-4763-87d2-3258d9640818/Untitled.png?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Content-Sha256=UNSIGNED-PAYLOAD&X-Amz-Credential=ASIAZI2LB466X5X7WXGN%2F20251204%2Fus-west-2%2Fs3%2Faws4_request&X-Amz-Date=20251204T164639Z&X-Amz-Expires=3600&X-Amz-Security-Token=IQoJb3JpZ2luX2VjEID%2F%2F%2F%2F%2F%2F%2F%2F%2F%2FwEaCXVzLXdlc3QtMiJIMEYCIQDy9%2BEA0ncCRkTtoaxtw0JQ7OTiFDHt1IUAofSEJMurjQIhAMxBbEGilb%2FSBr2Li
Amz-Signature=b5c0ea9c475392dc2cd8fb9e6a5450c733f76bd3fa39ec2c6ecfe5e1f4891c01&X-Amz-SignedHeaders=host&x-amz-checksum-mode=ENABLED&x-id=GetObject)

- **Props**

子组件可以通过 **props** 从父组件接受动态数据，向下傳遞到子組件

```
// ChildComp.vue
<template>
    <!-- 在模板中顯示 `msg` 屬性的值，如果未傳遞 msg 這個變數則顯示預設文字 -->
  <h2>{{ msg || 'No props passed yet' }}</h2>
</template>

<script setup>
// 使用 `defineProps` 定義一個 `props` 物件
const props = defineProps({
  msg: String
    // 定義 `msg` 屬性，並指定它的型別為字串，
    // 也可以用來指定標籤 props:['title'], template:`<h4>{{ title }}</h4>`
    // 這樣 App.vue 就可以設定成<blog-post title='...'></blog-post>
})
const props = defineProps({
  cover: {
      type: String,
  },
  tags: {
      type: Array,
  },
});
</script>

// ============================================
// App.vue
<template>
    <!-- 在模板中使用 ChildComp 組件，將 `greeting` 變數作為 prop 傳遞給子組件 -->
  <ChildComp :msg="greeting" />
</template>

<script setup>
import { ref } from 'vue'
import ChildComp from './ChildComp.vue'
```

```
const greeting = ref('Hello from parent')
</script>
```

- **Emits**

  **Emits **可以從子組件向父组件触发事件

```
// ChildComp.vue
<script setup>

<span style='color:orange'>// 使用 `defineEmits` 定義一個 `emit` 函式，
//「並指定允許發送 'response' 事件」</span>
const emit = defineEmits(['response'])

<span style='color:orange'>// 發送 'response' 事件，傳遞字串 'hello from child'</span>
emit('response', 'hello from child')
</script>

<template>
  <h2>Child component</h2>
</template>

// ===================================
// App.vue
<script setup>
import { ref } from 'vue'
import ChildComp from './ChildComp.vue'

const childMsg = ref('No child msg yet')
</script>

<template>
<span style='color:orange'> // 在模板中使用 ChildComp 組件，並監聽 'response' 事件，
  // 將子組件傳遞的訊息賦值給 `childMsg` 變數</span>
  <ChildComp @response="(msg) => childMsg = msg" />

<span style='color:orange'> <!-- 在模板中顯示子組件傳遞的訊息 --></span>
  <p>{{ childMsg }}</p>
</template>
```


 【查】ref, inject, provide

- Sort 名稱排序
  - 如果結果是負的，`a``b`

    被排序在

    之前。
  - 如果結果是正數，則`b``a`

    被排序在

    之前。
  - 如果結果是`0`

    ，兩個值的排序順序不做任何改變。

```plain text users.sort(function (a, b) { if (a.name < b.name) { return -1; } if (a.name > b.name) { return 1; } return 0; });
```

```python
const modelList = computed(() => {
    return models.value.sort((a, b) => {
        const nameA = a.modelName.toLowerCase();
        const nameB = b.modelName.toLowerCase();

        if (nameA < nameB) return -1;
        if (nameA > nameB) return 1;
        return 0;
    })
```

- this. 是一個特殊關鍵字，用於指向特殊實例

  舉例:this.$router.push('/catalog')

  通過 $router 物件調用 push 方法來導向 ('/catalog') 路徑

```
export default {
  name: "Links",
```

```
  methods:{
    catalog() { this.$router.push('/catalog'); },
    modelRequest() { this.$router.push('/test'); },
  }
};
```

- For 迴圈寫法

```
 const transformedOutputData = ref([]);
if (props.aboutProps.output.schema)
    for (const item of props.aboutProps.output.schema) {
        const transformedItem = {};
        for (const key in item) {
            const transformedKey = key.charAt(0).toUpperCase() + key.slice(1);
            transformedItem[transformedKey] = item[key];
        }
        transformedOutputData.value.push(transformedItem);
    }

// 與 python 差異不大，差異如下:
// 變數前面都需要先使用 const 或是 let
// in 改成 of
// : 該成

==============<其他案例>==================
const myFile = { file: yourFile1, video: yourFile2 };

Object.values(myFile).forEach((file) => {

    console.log(file);
});

============================
for (const s3FilePath of s3FilePathList) {
 console.log(s3FilePath)

============================
let val = {};

if (Array.isArray(inputTypesTest.value.url) && inputTypesTest.value.url.length === s3FilePathList.length) {
    for (let i = 0; i < inputTypesTest.value.url.length; i++) {
        val[inputTypesTest.value.url[i].name] = s3FilePathList[i];
    }
};
```

- 動態 CSS 寫法

    在標籤中除了可以寫入動態 CSS 外(:class=" ")，同時也可以加入靜態的(class=" ")

```
 // 【主程式中寫入定義好的 CSS 變數】
<div :class="[isFixedTabBar ? tabBarFixedClass : tabBarClass]">...<div>

// 【在 <script step> 中定義 CSS 變數】
<script step>
const tabBarFixedClass = ref('fixed top-36 space-x-2 bg-white w-full pt-10');
const tabBarClass = ref('flex pt-10');
```


- or || 符號可以常用，取代 v-if ； else

```
<div class="pt-2 text-gray-300 text-xs">
    {{ props.description || 'No description available.' }}
</div>

// 如果有資料就顯示 props.description，沒有的話就顯示 No description available.
```

- sort 寫法

```
 // 時間排序寫法
results.data = result.sort((a, b) => b.createdAt - a.createdAt);

// 英文名稱 A~Z 寫法
results.data = records.sort((a, b) =>
                a.modelName.localeCompare(b.modelName, undefined, {
                    sensitivity: 'base',
                })
            );
```

- Promise、Promise.all 的使用

```
const getUploadFilePath = async () => {
    const s3FilePathList = [];

    const uploadFileArray = Object.values(uploadFileObject.value);

    const uploadPromises = uploadFileArray.map(async (uploadFileValue) => {
    // 使用 map 對 uploadFileArray 內每個 uploadFileValue 執行操作。
        try {
             // 開始執行 API 呼叫與處理，這一段可以跳到下面
            let uploadPathRes = await getUploadPath(props.modelID);

            if (!uploadPathRes) throw new Error('');

            const s3FilePath = await uploadToS3(uploadFileValue);

            if (s3FilePath === '') throw new Error('');

            const datasetID = uploadPathRes.datasetID;
            const dir = uploadPathRes.dir;

            let datasetRes = await createDatasets({
                modelID: props.modelID,
                datasetID: datasetID,
                bucket: bucketName,
                dir: dir,
                datasetName: datasetID,
            });

            if (!datasetRes) throw new Error('');
                        // 如果是正確的，就會執行 push
            s3FilePathList.push(s3FilePath);
                        // 然後回傳一個 true
            return true;
        } catch (error) {
            return false;
        }
    });
        // 由於在 map 函數執行後，會產生一個包含多個 Promise 的陣列 uploadPromises
    // Promise.all 會等待所有的 Promise 完成，
    // 並返回一個包含所有 Promise 結果的新陣列 uploadTrue。
    const uploadTrue = await Promise.all(uploadPromises);
    // 使用 every 檢查 uploadTrue 陣列中的每個元素是否都等於 true
    const isUploadAllTrue = uploadTrue.every(
        (promiseResult) => promiseResult === true
    );

    return isUploadAllTrue ? s3FilePathList : [];
};
```

- async、await 非同步操作

    當功能為網絡請求、檔案讀取、計時器等會需要使用到非同步操作，因為不需要等待某些操作完成就可以繼續執行後續的代碼。這種操作的主要優勢是可以提高程式的效能和反應速度，因為它允許在等待某些操作完成時執行其他任務。

    如果在 function 中使用 async，JavaScript 解釋器(browsor 或是 node.js)就會推測 function 可能包含 await

    async、await 同時存在的時候，主要目的是確保你可以等待非同步操作完成，然後在你需要操作其結果的地方繼續執行代碼。這樣可以確保你不會在非同步操作完成之前試圖訪問或處理其結果，例如:我們要上傳多筆文件，然後將文件陸續儲存到一個陣列中，接著會處理這個陣列，所以我們需要使用 await 等所有文件上傳完畢並儲存到陣列中才，進行可以處理該陣列，如果沒有這一道程序，就會出現只有一筆資料被處理。

- m. c. 這樣符號的使用方法

    這是一個是用來呼叫不同的模組或物件的方法或函數的前綴

    - "m." 可能是指的是一個模型（Model）模組，用於處理數據庫操作或資料模型的相關邏輯。

    - "c." 可能是指的是一個控制器（Controller）模組，用於處理路由和請求處理邏輯。

    router.post('/model-filter-options', m.createCheck, c.createOption);

    - "m.createCheck" 可能是模型模組中的一個函數，用來執行某些數據庫操作或檢查。

    - "c.createOption" 可能是控制器模組中的一個函數，用來處理建立選項的相關邏輯。

- setTimeout 與 setInterval

    [https://kuro.tw/posts/2019/02/23/談談-JavaScript-的-setTimeout-與-setInterval/](https://kuro.tw/posts/2019/02/23/%E8%AB%87%E8%AB%87-JavaScript-%E7%9A%84-setTimeout-%E8%88%87-setInterval/)

    setTimeout( ) 延遲了某段時間 (單位為毫秒) 之後，才去執行「一次」

```
var timeoutID = window.setTimeout(( () => console.log("Hello!") ), 1000);

// 有如果希望在時間未到的時候就停止，可以使用 clearTimeout()
window.clearTimeout(timeoutID);
```

```
// 但如果 setTimeout() 的 callback function 已經被執行，那就等同是多餘的。
```

setInterval( ) 固定延遲了某段時間之後，才去執行對應的程式碼，然後「不斷循環」

```
var timeoutID = window.setInterval(( () => console.log("Hello!") ), 1000);

// 由於是不斷循環，所以需要使用 clearInterval() 停下來
window.clearInterval(timeoutID);
```

實際應用操作

```
題目:在五秒鐘之內，每秒鐘依序透過 console.log() 印出: 0 1 2 3 4 。」
for( var i = 0; i < 5; i++ ) {

  // 為了凸顯差異，我們將傳入後的參數改名為 x
  // 當然由於 scope 的不同，要繼續在內部沿用 i 這個變數名也是可以的。
  (function(x){
    window.setTimeout(function() {
      console.log(x);
    }, 1000 * x);
  })(i);
}
```

- filter 用來檢查是否符合條件

以下列程式中呈現，檢查每個模型 (`**model**`) 是否滿足特定的條件

```
models.value = models.value.filter((model) => {
        let include = false;

        if (selectedTag.length > 0) {
            selectedTag.forEach((tag) => {
                if (model.category.includes(tag)) include = true;
            });
        } else {
            include = true;
        }
        return include;
    });
```

- beforeEach(全局路由守衛) 監測 route 變化

```
// 假設今天我們使用 push 來改變路由
const goToLogin = () => {
    router.push('./login');
};
// 範例中有兩個路由
http://localhost:5173/seller/login
http://localhost:5173/login

// 望如果監測到 "seller" 就會顯示 true
const isSeller = ref(false);
router.beforeEach((to, from, next) => {
    isSeller.value = to.path.includes('seller');
    next();
});
```

beforeEach 是一個全局路由守衛，允許你在每次路由導航發生之前執行一些操作

以下是 `**beforeEach**` 的一些常見應用面向以及相關的操作範例:

- **權限驗證**:你可以使用 `**beforeEach**` 來檢查使用者是否有訪問特定頁面的權限。如果使用者未經授權，你可以將其導向登錄頁面或顯示錯誤信息。

```
router.beforeEach((to, from, next) => {
  const requiresAuth = to.meta.requiresAuth;
  const isAuthenticated = checkUserAuth(); // 假設有一個函數來檢查使用者是否已經登錄

  if (requiresAuth && !isAuthenticated) {
    next('/login'); // 未經授權，導向登錄頁面
  } else {
    next(); // 有權限，繼續導航
  }
});
```

- **修改路由行為**:你可以使用 `**beforeEach**` 來根據特定條件修改路由的行為，例如將某些路由重定向到其他路由。

```
router.beforeEach((to, from, next) => {
  if (to.path === '/old-route') {
    next('/new-route'); // 將 /old-route 重定向到 /new-route
  } else {
    next(); // 正常導航
  }
});
```

- **日誌記錄**:你可以使用 `**beforeEach**` 來記錄路由導航的紀錄，以便進行調試或分析。

```
router.beforeEach((to, from, next) => {
  console.log(`Navigating from ${from.path} to ${to.path}`);
  next(); // 正常導航
});
```

- **權限升級**:如果使用者在應用程序運行期間升級了權限，你可以使用 `**beforeEach**` 來更新路由的權限狀態。

```
router.beforeEach((to, from, next) => {
  if (userHasUpgradedPermissions()) {
    // 更新使用者的權限狀態
    updateUserData();

    // 繼續導航
    next();
  } else {
    next(); // 正常導航
  }
});
```

這些只是一些 `**beforeEach**` 的應用面向示例，你可以根據你的特定需求定義自己的路由守衛邏輯。`**beforeEach**` 提供了在路由導航前執行操作的強大能力，以幫助你更好地控制和管理你的應用程序的導航行為。

- localStorage 使用

這是一個類似 cookie 暫存檔的存在，可以儲存進去 localhost 中，達到全域的資料共享，但是會因為瀏覽器的關閉，暫存也會消失，所以需要特別注意使用

- Promise 處理異步操作標準方法

使用 .then、.catch

- .then 方法

當 Promise 成功時會回調一個參數，這時候由 .then 接手處理後續的邏輯操作

```
const promise = new Promise((resolve, reject) => {
  // 异步操作，最终会调用 resolve 或 reject
});

promise.then(result => {
  // 在 Promise 解决时调用，result 是解决时的结果
}).catch(error => {
  // 在 Promise 拒绝时调用，error 是拒绝时的错误信息
});
```

- .catch 方法

當 Promise 回調參數失敗，這時候由 .catch 接收錯誤訊息參數

```
const promise = new Promise((resolve, reject) => {
  // 异步操作，最终会调用 resolve 或 reject
});

promise.then(result => {
  // 在 Promise 解决时调用
}).catch(error => {
  // 在 Promise 被拒绝时调用，error 是拒绝时的错误信息
});
```

- addEventListener 監聽事件

以監聽及時滾輪位置為例

```
import { onMounted, onBeforeUnmount } from 'vue';

function handleScroll() {
  const scrollY = window.scrollY;
  console.log(scrollY);
}
```

```
onMounted(() => {
  window.addEventListener('scroll', handleScroll);
});

onBeforeUnmount(() => {
  window.removeEventListener('scroll', handleScroll);
});

// 這邊要注意，使用 addEventListener 後
```

```
    - addEventListener 其他應用
```

```
 // **Click Event:**
document.getElementById('myButton').addEventListener('click', handleClick);

// **Keyboard Event:**
document.addEventListener('keydown', handleKeyPress);

// **Mouse Move Event:**
document.addEventListener('mousemove', handleMouseMove);

// **Form Event:**
document.getElementById('myForm').addEventListener('submit', handleFormSubmit);

// **Window Event:**
window.addEventListener('resize', handleResize);

// **Custom Event:**
const customEvent = new Event('myCustomEvent');
document.addEventListener('myCustomEvent', handleCustomEvent);
```

```
        <br/>

        <br/>

        <br/>

        <br/>

        <br/>

        <br/>

        <br/>

        <br/>
```

```
- 功能有兩種寫法，函數聲明 **function handleScroll() {}**、箭頭函數 **const handleScroll = () => {}**

    兩種寫法都可以，主要是會需要看需求以及編碼風格
```

```
// 函數聲明
function add(a, b) {
  return a + b;
}

// 箭頭函數
const add = (a, b) => a + b;
```

```
    <br/>

    <br/>

- computed 即時數據監聽，並即時重新”計算”其值
```

```
const a = computed(() => { return useStore.names })

// computed 函數的作用是創建一個計算屬性 a，
// 其值基於 useStore.names 的值動態計算，並在 useStore.names 發生變化時自動更新。
// 這有助於實現數據的響應式更新。
// 所以當 useStore.names 的值發生變化時，a 會即時重新計算，
// 這有助於簡化程式碼並確保數據的一致性。
```

```
    <br/>

- watchEffect 監聽相應數據變化

    這個 function 用於執行某些副作用操作，例如發送 API 請求、更新 DOM、觸發事件等，當监视的响应式数据发生变化时，這些操作會被触发。這對於需要<span style='color:orange'>自動反應数据变化</span>的操作非常有用。

- computed  VS watchEffect
```

1. **使用 **`**computed**`:

   - 使用 `**computed**` 來派生新的值或計算屬性，這些值依賴於一個或多個响应式数据，並且在計算過程中不會對應用程序的狀態造成變化。

   - 用 `**computed**` 來處理需要經常計算的數值，例如平均值、總和、過濾、排序等操作。

1. **使用 **`**watchEffect**`:

   - 使用 `**watchEffect**` 來處理副作用操作，這些操作可能會影響應用程序的狀態，例如發送 API 請求、更新 DOM、設置本地存儲、觸發動畫等。

   - 使用 `**watchEffect**` 當你需要在响应式数据發生變化時自動執行操作，而不必關心特定的派生值。

`**watchEffect**` 和 `**computed**` 都用於處理响应式数据，但它們之間存在一些重要的差異

- 1. **執行時機**:

   - `**computed**`:計算屬性是在响应式数据的依賴发生变化时進行計算的。當其依賴的响应式数据發生變化時，`**computed**` 會自動重新計算並返回新值。

   - `**watchEffect**`:`**watchEffect**` 會立即執行並監視整個函數內部，當函數內部的任何响应式数据發生變化時，它會自動執行函數。

- 2. **返回值**:

   - `**computed**`:它返回一個新的响应式数据，這個数据的值是由計算函數返回的值。這個計算屬性可以像普通的响应式数据一樣被訪問。

   - `**watchEffect**`:它不返回值，而是通常用於執行副作用操作，例如發送 API 請求或更新 DOM，而不返回一个新的响应式数据。

- 3. **依賴聲明**:

   - `**computed**`:你需要明確聲明計算屬性的依賴，Vue 會追蹤這些依賴，並在它們變化時重新計算計算屬性。

   - `**watchEffect**`:它會自動追蹤函數內部使用的所有响应式数据，不需要明確聲明依賴，這使代碼更簡潔，但可能會造成效能開銷，因為它監視了更多的變數。

- 4. **使用情境**:

   - `**computed**`:適用於需要根據一組响应式依賴計算派生值的情況，例如過濾、排序、格式化数据等。

   - `**watchEffect**`:適用於需要執行副作用操作的情況，例如發送網路請求、設置本地存儲、觸發動畫等。

總之，`**computed**` 用於派生新的响应式数据，並需要明確聲明依賴，而 `**watchEffect**` 用於執行副作用操作，它不需要明確聲明依賴並且是一個更強大的工具，但應謹慎使用以避免不必要的效能開銷。

- 使用實際的程式碼情境(計算購物車的總價格)來比較會更清楚:

computed 情境中，`**totalAmount**` 是一個 `**computed**` 屬性，它**依賴**於 `**cartItems**` 和 `**products**` 這兩個响应式数据。每當 `**cartItems**` 或 `**products**` 中的數據發生變化時，`**totalAmount**` 會自動重新計算購物車的總價格。

```
import { ref, computed } from 'vue';

const cartItems = ref([]); // 購物車的商品列表
const products = ref([]); // 所有商品列表

// 創建一個計算屬性，計算購物車的總價格
const totalAmount = computed(() => {
  let total = 0;
  for (const item of cartItems.value) {
    const product = products.value.find(p => p.id === item.productId);
    if (product) {
      total += product.price * item.quantity;
    }
  }
  return total;
});
```

watchEffect 情境中

```
import { ref, watchEffect } from 'vue';

const cartItems = ref([]); // 購物車的商品列表
const products = ref([]); // 所有商品列表
const totalAmount = ref(0); // 總價格

// 使用 watchEffect 執行副作用操作
watchEffect(() => {
  totalAmount.value = 0;
  for (const item of cartItems.value) {
    const product = products.value.find(p => p.id === item.productId);
    if (product) {
      totalAmount.value += product.price * item.quantity;
    }
  }
});
```

我們使用 `**watchEffect**` 直接執行副作用操作，而不是像 `**computed**` 那樣返回一個新的响应式数据。這將導致每次 `**cartItems**` 或 `**products**` 發生變化時，`**totalAmount**` 會被重新計算。

儘管 `**watchEffect**` 可以實現相同的功能，但它不像 `**computed**` 那麼清晰和易於管理。`**computed**` 專注於計算，並且具有自動緩存，這意味著當依賴沒有變化時，它不會重複計算，這有助於提高性能。因此，通常情況下，對於派生值的計算，`**computed**` 更適合，而 `**watchEffect**` 則更適合執行副作用操作，如發送 API 請求或更新 DOM。

- 透過 id 找尋 Array 中的物件 targetArray.find((obj) => obj.id === id)

```
// 資料結構
const purchasesOnPremRows = ref([
    {
      id: '123',
      name: '',
      accessCode: '',
      purchaseDate: 'Oct 26, 2023',
      pricingPlans: 'Per instance',
      hasAccessCode: false,
      hasClickGetAccessButton: false,
    },
    {
      id: '234',
      name: '',
      accessCode: 'r4_QqAxxxxxxxxxxxx',
      purchaseDate: 'Oct 23, 2023',
      pricingPlans: 'Per instance',
      hasAccessCode: true,
      hasClickGetAccessButton: true,
    },
]

const result = purchasesOnPremRows.find((obj) => obj.id === '234')

console.log(result)
// 就可以拿到下方資料
    {
      id: '234',
      name: '',
      accessCode: 'r4_QqAxxxxxxxxxxxx',
      purchaseDate: 'Oct 23, 2023',
      pricingPlans: 'Per instance',
      hasAccessCode: true,
      hasClickGetAccessButton: true,
    },
```

- 透過 特徵 找尋 Array 中的多個物件 targetArray.filter((obj) => obj.id === id)

```
// 資料結構
const purchasesOnPremRows = ref([
    {
      id: '123',
      name: '',
      accessCode: '',
      purchaseDate: 'Oct 26, 2023',
      pricingPlans: 'Per instance',
      hasAccessCode: false,
      hasClickGetAccessButton: false,
    },
    {
      id: '234',
      name: '',
      accessCode: 'r4_QqAxxxxxxxxxxxx',
      purchaseDate: 'Oct 23, 2023',
      pricingPlans: 'Per instance',
      hasAccessCode: true,
      hasClickGetAccessButton: true,
    },
]

const result = purchasesOnPremRows.filter((obj) => obj.hasAccessCode === true)

console.log(result)
// 就可以拿到下方資料，如果有多個一樣符合 hasAccessCode === true 就會被找出來
    {
      id: '234',
      name: '',
      accessCode: 'r4_QqAxxxxxxxxxxxx',
      purchaseDate: 'Oct 23, 2023',
      pricingPlans: 'Per instance',
      hasAccessCode: true,
      hasClickGetAccessButton: true,
    },
```

- 透過尋找

- 透過 map 對應

```
const purchasesOnPremRows = ref(
    [
      {
        id: '123',
        name: '',
        accessCode: '',
        purchaseDate: 'Oct 26, 2023',
        pricingPlans: 'Per instance',
        hasAccessCode: false,
        hasClickGetAccessButton: false,
      },
      {
        id: '234',
        name: '',
        accessCode: 'r4_QqAxxxxxxxxxxxx',
        purchaseDate: 'Oct 23, 2023',
        pricingPlans: 'Per instance',
        hasAccessCode: true,
        hasClickGetAccessButton: true,
      },
].map((item) => {
      item.hasAccessCode = item.accessCode !== '';
      return item;
    })
  );

// 假設如果發現 accessCode 有數值那就是 true 那就將 true，帶入 hasAccessCode
```

Vuex 數據儲存

- 【基礎操作】套件安裝、註冊

    - 一、安裝套件

```
npm install vuex
npm install vuex-persistedstate
```

    - 二、設定 main.js

```
import store from './store';

const vueApp = createApp(App)
    .use(store)
    ...

vueApp.mount('#app');
```

    - 三、建立全域變數 JS 檔(可自行增加其他變數或是初始數據)

```
// src\store\modules\globalContants.js
// state（data）響應式的資料狀態儲存, 資料狀態變化時，有用到的 component 都會即時更新
const state = {
  globalConstants: {
    isDialogFilterOpen: false,
  },
};

// getter（computed）
// 1.加工資料呈現
// 2.同 computed 一樣會被緩存、依賴值變更了才會重新計算
const getters = {
  getDialogOpenState: (state) => state.globalConstants.isDialogFilterOpen,
};

// action（methods）
// 1.操作同步或異步事件的處理但不直接修改資料（state）
// 2.是透過commit → 呼叫 mutation 改變 state
// Get WebAdm 1-6 Emmission Data
const actions = {
  UpdateConstants(
    {
      commit, //, rootState
    },
    model
  ) {
    //保留
    commit('globalConstants', model);
```

```
  },
};

// mutation
// 1.改變state
// 2.只處理同步函數:不要在這進行非同步的動作（例如 setTimeout / 打API取遠端資料...等）
// 3.mutations must be synchronous.
const mutations = {
  GlobalConstants(state, data) {
    //儲放資料
    state.globalConstants = data;
  },
};

export default {
  state,
  getters,
  actions,
  mutations,
};
```

- 四、建立啟用入口的 JS 檔

```
// src\store\index.js
import Vuex from 'vuex';
import createPersistedState from 'vuex-persistedstate';

import globalConstants from './modules/globalContants.js';

// Create store
export default new Vuex.Store({
  strict: true,
  modules: {
    globalConstants,
  },
  plugins: [createPersistedState()],
});
```


- 【數據使用】

- 五、使用方法

- 使用情境一、在 first.vue 中取用全域變數 globalContants.js 內的 dialog 開關

```
<template>
    <div class="col-1">{{ isDialogFilterOpen }}</div>
</template>
export default {
    name: 'First',
  data() {
      isDialogFilterOpen: this.$store.getters.getDialogOpenState,
  }
}
```

- 使用情境二、在 first.vue 中更新 dialog 開關，而 second.vue 的 dialog 開關也會同步變更

建立 Vue 專案 `!!! 常用 !!!`

- 專案基礎建設

在 Terminal 中，寫入「npm create vite@latest」就會進到安裝流程，注意，如果 Select a variant 的時候，沒有選客製，那就會需要額外手動處理，Pinia、ESLint、Prettier，所以建議選擇「Customize with create-vue」但這個不是每次都有，因此會需要特別注意！

- CSS 套件引入

完成後，加入 `vuetify` CSS 庫，可提升後續的 RWD 與 CSS 的操作

```
npm install vuetify@latest
// vuetify 相依性檔案
npm install vuetify @mdi/font @mdi/js
npm i vite-plugin-vuetify
```

- 建立 src\plugins\vuetify.ts 檔案

```
// src/plugins/vuetify.ts
import { createVuetify } from 'vuetify'
import 'vuetify/styles' // 匯入 Vuetify 樣式
import { aliases, mdi } from 'vuetify/iconsets/mdi-svg' // 使用 Material Design Icons

// 若需要進一步自訂 Vuetify 主題，請在這裡配置
const vuetify = createVuetify({
  icons: {
    defaultSet: 'mdi',
    aliases,
    sets: {
      mdi,
    },
  },
  theme: {
    defaultTheme: 'light',
    themes: {
      light: {
        colors: {
          primary: '#1976D2',
          secondary: '#424242',
          accent: '#82B1FF',
          error: '#FF5252',
          info: '#2196F3',
          success: '#4CAF50',
          warning: '#FB8C00',
        },
      },
    },
  },
})

export default vuetify
```

- 調整 src\main.ts 檔案 - 引入 vuetify.ts

```
import './assets/main.css'

import { createApp } from 'vue'
import { createPinia } from 'pinia'

import App from './App.vue'
import router from './router/router' // 匯入 router 而不是 routes
import vuetify from './plugins/vuetify'

import 'vuetify/styles' // 引入 Vuetify 樣式

const app = createApp(App)

app.use(createPinia())
app.use(router) // 使用 router 實例
app.use(vuetify)

app.mount('#app')
```

- 調整 .\vite.config.ts 檔案

```
import { fileURLToPath, URL } from 'node:url'

import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'
import vueDevTools from 'vite-plugin-vue-devtools'

import vuetify from 'vite-plugin-vuetify'

// https://vite.dev/config/
export default defineConfig({
  plugins: [
    vue(),
    vueDevTools(),
    vuetify({
      autoImport: true, // 啟用自動引入
```

```
    }),
  ],
  resolve: {
    alias: {
      '@': fileURLToPath(new URL('./src', import.meta.url))
    }
  }
})
```

- 基礎專案框架建立

    - 建立 RWD 設定

        參考文章(斷點):https://ithelp.ithome.com.tw/articles/10325157?sc=rss.iron

        效果呈現:https://codepen.io/Andy-Chen/pen/vmaqxP

    - 建立 views

        這裡主要放 Router 切換過去的 .vue 頁面程式碼

    - 建立 components

        這裡是每個頁面會用的組合 component 程式以及常用的 component 程式

    - 建立 layout

        這裡可以建立 header、footer 以及不同的 layout 模板

    - 設定 router

        這裡設定 Router，並可以透過 pinia 內存的權限資訊 (API 傳入的權限存進 pinia)，管理那些帳號可以進去哪些區域

    - 設定 App

        設定甚麼 router 使用什麼 layout

Vite

- 介紹

    ![Untitled.png](https://prod-files-secure.s3.us-west-2.amazonaws.com/87b495d8-0bc3-49b6-9777-8698fe91b622/6dc70bc8-2dc1-4186-9bf7-9c0b081e37c9/Untitled.png?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Content-Sha256=UNSIGNED-PAYLOAD&X-Amz-Credential=ASIAZI2LB4665T7FOZ51%2F20251204%2Fus-west-2%2Fs3%2Faws4_request&X-Amz-Date=20251204T164616Z&X-Amz-Expires=3600&X-Amz-Security-Token=IQoJb3JpZ2luX2VjEID%2F%2F%2F%2F%2F%2F%2F%2F%2F%2FwEaCXVzLXdlc3QtMiJIMEYCIQCOrVDNr%2F64N%2F2eGaenmBXdfkutxpXAIPYvwuZG5fQXPgIhALp3mLZb2qj%2BMWPAmz-Signature=60db61f9405a79a45779a340d21694f23f7e685e7b598dd468c35c28c7a6e597&X-Amz-SignedHeaders=host&x-amz-checksum-mode=ENABLED&x-id=GetObject)

    Vite 與以往的傳統打包方式不同，透過利用瀏覽器支援的 Native ESM 進行運作

    Vite 根據 Http 的 request 來載入模組進行處理，實現真正的依需求加載。

- 啟動 vite 專案指令(雙開)

    一般來說啟動專案會使用「npm run dev」然後預設 localhost:5173 就會啟動，但如果要雙開，就可以使用「npx vite --port 5174」會開啟另外一個 port 號使用

- Slot 插槽 (分為三種，默認、具名、作用域插槽)

    我的看法是，這個操作可以解決之前我們 header 資訊切換的問題，可以使用 name 進行切換 for seller / buyer 的 header 資訊

    - 默認插槽的使用方式 <slot></slot>

```
// App.vue
<template>
        <div id='app'>
                <HelloWorld>這是默認插槽</HelloWorld>
        </div>
</template>

// * 這邊提供不同寫法，
// 具名寫法 <template #center> ，默認寫法 <template #default>


// HelloWorld.vue
```

```
<template>
        <div class='hello'>
                <slot></slot>
        </div>
</template>

====================================
// 結果
<slot></slot> 的位置，就會替換成 "這是默認插槽" 的字樣
```

- 具名插槽的使用方式 <slot name='center'></slot>

```
 // App.vue
<template>
        <div id='app'>
                <HelloWorld>這是默認插槽
                        <template v-slot:center>這是具名插槽</template>
                </HelloWorld>
        </div>
</template>

// * 這邊提供不同寫法，
// 具名寫法 <template #center> ，默認寫法 <template #default>

// HelloWorld.vue
<template>
        <div class='hello'>
                <slot name='center'></slot>
        </div>
</template>

====================================
// 結果
<slot name='center'></slot> 的位置，就會替換成 "這是具名插槽" 的字樣
```

- 作用域插槽的使用方式 (將子組件傳遞給父組件)

```
 // App.vue
<template>
        <div id='app'>
                <HelloWorld>這是默認插槽
                        <template v-slot:center>這是具名插槽</template>
                        <template v-slot:bottom='slotProp'>{{slotProp.sayHi}}</template>
                </HelloWorld>
        </div>
</template>


// HelloWorld.vue
<template>
        <div class='hello'>
                <slot :sayHi:'sayHi' name='bottom'></slot>
        </div>
</template>
<script setup>
const data = ref({sayHi:'大家好，我是作用插槽'})
</script >

====================================
// 結果
<slot :sayHi:'sayHi' name='bottom'></slot> 的位置，
就會替換成 "大家好，我是作用插槽" 的字樣
```


Quasar Framework

- 介紹

  Quasar Framework 是一個基於 Vue.js 的前端開源框架， 並提供豐富的 Component 樣式，包含表單、對話框、按鈕、導航欄位…

  目前專案雖然是使用 Vue 開發，然後結合 Quasar 的組件工具架設，但其實 Quasar 可以獨立創建專案

  Quasar Framework 除了支援 Web 應用，還可以用來開發 App、桌面應用程式

- 切版 layout、container、page

  1. `**<q-layout>**`:Quasar中的佈局容器，通常包含整個頁面的佈局結構。

  1. `**<q-page-container>**`:頁面容器，用於包含頁面的內容，可以擁有多個`**<q-page>**`。

  1. `**<q-page>**`:表示一個頁面，通常包含了頁面的主要內容。

    1. `**<q-header>**`:頁面頂部的容器，通常包含標題、導航欄等元素。

    1. `**<q-footer>**`:頁面底部的容器，通常包含頁腳信息。

    1. `**<q-toolbar>**`:工具欄容器，通常用於放置工具按鈕、選項卡等。

    1. `**<q-btn>**`:按鈕元素，用於創建交互式按鈕。

    1. `**<q-input>**`:輸入框元素，用於接收用戶輸入。

    1. `**<q-card>**`:卡片元素，用於呈現信息區塊。

    1. `**<q-list>**`:列表元素，用於顯示項目列表。

- 教學討論空間

    這個帳號有一些 Quasar 的應用，可以用來修改我們的程式 https://codepen.io/turigeza

RWD

以 Navbar 的狀態使用為例

會先定義 style 的方式，以此切換成不同裝置尺寸下呈現的 element

- <style> code

```
// ref:seller-draft-bar.vue

<style lang="sass" scoped>
.desktop-device
  display: none
.mobile-device
  display: flex

@media (min-width: 768px)
  .desktop-device
    display: flex
  .mobile-device
    display: none
</style>

// 解釋
// 一開始定義兩個 class 屬性 .desktop-device、.mobile-device
// 而 .desktop-device 設為 none (不顯示)，.mobile-device 為 flex

// 接著使用「媒體查詢」的方式，重新定義為，當裝置框度大於或等於 768px 的時候
// .desktop-device 設為 flex，.mobile-device 為 none (不顯示)

// scoped 解釋
// 這是一個用來鎖定該檔案才使用的 style，不會讓後代全局使用到這個 style

// 關於 scoped 還有一個用法，就是加入 :deep()
// :deep() 舉例:
<style scoped>
:deep(h1) {
  color: darksalmon;
}
</style>
// 表示所有後代元件的 <h1> 都會被這個樣式影響到
```


- 延伸功能 scopped、:deep()、:slotted()

    ref: https://ithelp.ithome.com.tw/articles/10300975

    - 【scoped】這是一個用來鎖定該檔案才使用的 style，不會讓後代全局使用到這個 style

        不加 scoped，子元件也會吃到 style

        ![Untitled.png](https://prod-files-secure.s3.us-west-2.amazonaws.com/87b495d8-0bc3-49b6-9777-8698fe91b622/521b3af8-696d-468a-8b53-91f7859c5868/Untitled.png?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Content-Sha256=UNSIGNED-PAYLOAD&X-Amz-Credential=ASIAZI2LB466ZPS6GHHR%2F20251204%2Fus-west-2%2Fs3%2Faws4_request&X-Amz-Date=20251204T164616Z&X-Amz-Expires=3600&X-Amz-Security-Token=IQoJb3JpZ2luX2VjEID%2F%2F%2F%2F%2F%2F%2F%2F%2F%2FwEaCXVzLXdlc3QtMiJIMEYCIQDP1YWQFrDjZmzuDD3pIjc5eMCXNByuqd3fBgBjUZ0ndgIhAKFcBX2clyX6PgtdACI%2Amz-Signature=40cdcb2f54e88f48633186a314a63b8c8eda4caaf677d22fa880fe10f0ec1937&X-Amz-SignedHeaders=host&x-amz-checksum-mode=ENABLED&x-id=GetObject)

        加入 scoped，可以避免子元件也會吃到 style

        ![Untitled.png](https://prod-files-secure.s3.us-west-2.amazonaws.com/87b495d8-0bc3-49b6-9777-8698fe91b622/a5df985e-ef70-461e-9368-83ae3e0a194a/Untitled.png?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Content-Sha256=UNSIGNED-PAYLOAD&X-Amz-

Credential=ASIAZI2LB466ZPS6GHHR%2F20251204%2Fus-west-2%2Fs3%2Faws4_request&X-Amz-Date=20251204T164616Z&X-Amz-Expires=3600&X-Amz-Security-Token=IQoJb3JpZ2luX2VjEID%2F%2F%2F%2F%2F%2F%2F%2F%2F%2FwEaCXVzLXdlc3QtMiJIMEYCIQDP1YWQFrDjZmzuDD3pIjc5eMCXNByuqd3fBgBjUZ0ndgIhAKFcBX2clyX6PgtdACI%2Amz-Signature=1ff706a578e08f7e6f5a66df456094fa63f596f0dd244d0e96bb3e8545138d67&X-Amz-SignedHeaders=host&x-amz-checksum-mode=ENABLED&x-id=GetObject)

- 【:deep( )】如果希望該元件後代也可以使用，就會使用

```
// :deep() 舉例:
<style scoped>
:deep(h1) {
  color: darksalmon;
}
</style>
// 表示所有後代元件的 <h1> 都會被這個樣式影響到
```

- 【:slotted( )】

使用上會使用動態 class 「:class="deviceClass"」綁定到指定的 element，然後再 <script> 內使用 props 的方式進行狀態切換

- <template><script setup> code

```
// ref:seller-draft-bar.vue

<template>
<div
    :class="deviceClass"
    class="justify-center w-full bg-white border-bottom"
  >
    <div ...
<template>

<script setup lang="ts">
import { computed } from 'vue';

type Props = {
  device: 'desktop' | 'mobile';
}
const props = withDefaults(defineProps<Props>(), {
  device: 'mobile'
});

const emit = defineEmits<{
  (e: 'update:modelValue', value: string): void;
}>();

const deviceClass = computed(()=>{
  if(props.device === 'desktop'){
    return 'desktop-device'
  }else if(props.device === 'mobile'){
    return 'mobile-device'
  }
  return undefined
})
</script>

// 除了這個方法，
// 也可以使用 <my-component :device="'desktop'"></my-component> 的方式，
// 進行component 的切換
```

API 串接

```
1. 前置作業:安裝axios 「npm install axios」

1. 創建API服務:在`**src**`文件夾中創建一個新的`**services**`文件夾，並在其中創建一個`**api.js**`文件，

    - 程式範例如下:`**baseURL**`為實際API URL
```

```
// src/services/api.js
import axios from 'axios';

const baseURL = 'https://api.example.com'; // 修改為你的API基本URL

const api = axios.create({
  baseURL,
  timeout: 10000, // 請求超時時間（毫秒）
});
// axios庫的create方法來創建一個自定義的API實例，用來發送HTTP請求
// 如果呼叫的時間超過 10000 毫秒則會被取消，避免應用程序被阻塞

export default api;
```

1. 在 Vue 組件中，請求，引入創建的API服務文件，

1. 如果有多個 API 串接在同一個網頁上，則會需要再 ./src/services files 內建立多個 api.js，然後網頁中的 method 建立多個 API 端口

    - 範例如下:

```
<template>
  <div>
    <button @click="fetchDataFromApi1">從API1獲取數據</button>
    <button @click="fetchDataFromApi2">從API2獲取數據</button>
  </div>
</template>

<script>
import api1 from '@/services/api1';
import api2 from '@/services/api2';

export default {
  methods: {
    async fetchDataFromApi1() {
      try {
        const response = await api1.get('/endpoint');
        console.log('Data from API1:', response.data);
      } catch (error) {
        console.error('Error:', error);
      }
    },
    async fetchDataFromApi2() {
      try {
        const response = await api2.get('/endpoint');
        console.log('Data from API2:', response.data);
      } catch (error) {
        console.error('Error:', error);
      }
    },
  },
};
</script>
```

1. 使用習慣建議

    - API 名稱習慣

```
const DX_API = {
    MODEL_LIST: `${DX_API_SERVER}/models`,
    MODEL_INFO: `${DX_API_SERVER}/models:modelID`,
}

// 使用在 Vue 頁面的時候，就可以使用 replace 的方式來 Call
${DX_API_SERVER}/models:modelID`.replace(':modelID', modelID)
// 雖然可以使用下面的方式來 call API 但是在最上面的 API 管理庫就會很亂
${DX_API_SERVER}/models + modelID

192.168.0.4:4000
```

```
        寫 v-for 需要習慣使用 key

        <br/>

        <br/>

        <br/>

        <br/>
```

- API 撰寫案例實際說明

```
const getRequestDetail = async (modelRequestID) => {
// 使用 async 開始異步操作，透過 async 可以在函數內使用 await
// 來等待 Promise 處理異步操作後回傳的參數
    const output = await axios.get(DX_API.REQUEST_DETAIL.replace(':modelRequestID', modelRequestID))
// 使用 axios 發起 HTTP 請求，獲得 response 物件(可自行命名)
    .then(response => {
        if (response.status !== StatusCodes.OK) {
                        // throw 用來拋出一個異常
                        // new Error() 表示創建一個新的 Error 對象
                        // response.data 是一個錯誤內容，通常會是一個字串，描述錯誤發生內容
                        // 如果執行到 throw 會中斷當下程式碼作業流程，並傳到 .catch() 進行錯誤處理
            throw new Error(response.data);
            }
        return response.data;
    })
        // 如果物件的狀態是 200
```

```
    //接著，將物件的數據 response.data 存到 data 變數中，並回傳出來
    .then(data => {
        return data.data;
    })
        // 如果物件的狀態是 error
        // 那就會回傳錯誤訊息
    .catch(error => {
        console.log(error)
        // TODO
        return {};
    })

return output;
}
```

- API 串接實例 (資料傳送到後端)

    - API 文件(sendRequestInfo):

```
const sendRequestInfo = async (requestData) => {
    const output = await axios.post(DX_API.MODEL_REQUEST, requestData)
        .then(response => {
            if (response.status !== StatusCodes.CREATED) {
                throw new Error(response.data);
            }

            return response.data;
        })
        .then(data => {
            return data.data;
        })
        .catch(error => {
            console.log(error)
            return [];
        })

    return output;
}
```

    - 主程式:

```
<template>
    <div class="md:container md:mx-auto px-32 py-10">
        <div class="">
            <button
                type="button"
                @click="back"
                class="py-3 px-4 inline-flex justify-center items-center gap-2 rounded-md border border-transparent font-semibold text-gray-500
hover:text-gray-700 focus:outline-none focus:ring-2 ring-offset-white focus:ring-gray-500 focus:ring-offset-2 transition-all text-sm"
            >
                {{ '< Model Request' }}
            </button>
        </div>

        <!-- Name -->
        <div class="pt-10">
            <div
                v-for="question in baseQuestionList"
                :key="'b-' + question.id"
                class="pt-4"
            >
                <label
                    for="hs-trailing-icon"
                    class="block font-semibold text-2xl mb-2"
                >
                    {{ question.title }}
                </label>
                <label
                    for="hs-trailing-icon"
                    class="text-gray-800 mb-2 font-normal text-base pt-4"
                >
                    {{ question.notice }}
                </label>
                <div class="pt-4"></div>
                <div class="relative">
                    <input
                        type="text"
                        v-model="baseInputs[question.id]"
                        class="border py-3 px-4 block w-full bg-white border-gray-200 rounded-md text-sm focus:border-blue-500 focus:ring-blue-
500"
                        :class="[
                            hasClickSubmitBtn === true &&
                            question.required &&
                            baseInputs[question.id] === ''
                                ? 'border-red-500'
```

```
                            : 'border-gray-200',
                    ]"
                    :placeholder="question.placeholder"
                />

                <div
                    v-if="
                        hasClickSubmitBtn &&
                        question.required &&
                        baseInputs[question.id] === ''
                    "
                    class="absolute inset-y-0 right-0 flex items-center pointer-events-none pr-3"
                >
                    <svg
                        class="h-5 w-5 text-red-500"
                        width="16"
                        height="16"
                        fill="currentColor"
                        viewBox="0 0 16 16"
                        aria-hidden="true"
                    >
                        <path
                            d="M16 8A8 8 0 1 1 0 8a8 8 0 0 1 16 0zM8 4a.905.905 0 0 0-.9.995l.35 3.507a.552.552 0 0 0 1.1 0l.35-
3.507A.905.905 0 0 0 8 4zm.002 6a1 1 0 1 0 0 2 1 1 0 0 0 0-2z"
                        />
                    </svg>
                </div>
            </div>
            <div class="pt-2"></div>
            <div
                v-if="
                    hasClickSubmitBtn &&
                    question.required &&
                    baseInputs[question.id] === ''
                "
                class="flex items-center pointer-events-none"
            >
                <span class="text-red-400 font-normal text-sm"
                    >Please enter content.</span
                >
            </div>
        </div>
    </div>

    <!-- submit button -->
    <div class="pt-32 pb-10">
        <button
            type="button"
            @click="handleSubmit"
            class="py-3 px-4 inline-flex justify-center items-center gap-2 rounded-md border font-semibold text-sm bg-yellow-600 text-gray-
50 shadow-sm align-middle hover:bg-yellow-700 focus:outline-none focus:ring-2 focus:ring-offset-2 focus:ring-offset-white focus:ring-yellow-700
transition-all float-right"
        >
            Sumit the model request
        </button>
        <div v-if="showSuccessMessage" class="text-yellow-600">
            Submitted Successfully!
        </div>
    </div>
</div>
</template>

<script setup>
import { ref } from 'vue';
import { useRouter } from 'vue-router';

import modeRequestData from '../configs/model-request-submit.js';
import { sendRequestInfo } from '../utils/dx-api.js';

const router = useRouter();

const baseQuestionList = ref(modeRequestData.baseQuestionList);
const descriptionQuestionList = ref(modeRequestData.descriptionQuestionList);
const requirementQuestionList = ref(modeRequestData.requirementQuestionList);
const personalQuestionList = ref(modeRequestData.personalQuestionList);

const baseInputs = ref({
    requestName: '',
});

const descriptionInputs = ref({
    description: '',
});

const requirementInputs = ref({
    industry: '',
    problem: '',
    keyFunctionalities: '',
```

```
    inoutDescription: '',
    excepting: '',
});

const personalInputs = ref({
    name: '',
    email: '',
    company: '',
    role: '',
});

// Determine whether the submit button has been pressed to trigger the
// display of CSS for unfilled fields.
const hasClickSubmitBtn = ref(false);

/**
 * Check if all the required fields have input values
 *
 * @param {*} questionList
 * @param {*} inputList
 */
const emptyInputCheck = (questionList, inputList) => {
    for (let index = 0; index < questionList.length; index++) {
        const q = questionList[index];

        if (inputList[q.id] === '' && q.required) {
            return true;
        }
    }
    return false;
};

// 將 baseQuestionList、baseInputs 的數值套用到 emptyInputCheck 內
const handleSubmit = () => {
    hasClickSubmitBtn.value = true;

    const isBaseQuestionEmpty = emptyInputCheck(
        baseQuestionList.value,
        baseInputs.value
    );
    const isDescriptionEmpty = emptyInputCheck(
        descriptionQuestionList.value,
        descriptionInputs.value
    );
    const isRequirementQuestionEmpty = emptyInputCheck(
        requirementQuestionList.value,
        requirementInputs.value
    );
    const isPersonalQuestionEmpty = emptyInputCheck(
        personalQuestionList.value,
        personalInputs.value
    );
// 只要出現空值就會跳出請輸入的訊息
    if (
        isBaseQuestionEmpty ||
        isDescriptionEmpty ||
        isPersonalQuestionEmpty ||
        isRequirementQuestionEmpty
    ) {
        alert('Please enter input');
        return;
    }
// else 就會將數值整理成一個
    const requestData = {
        ...baseInputs.value,
        ...descriptionInputs.value,
        ...requirementInputs.value,
        ...personalInputs.value,
    };

    sendRequestInfo(requestData)
        .then((response) => {
            alert('Submission Successful');
        })
        .catch((error) => {
            console.error('An Error Occurred', error);
            alert('An Error Occurred');
        });
};

const back = () => {
    router.push('/test');
};
</script>
```

- API 串接實例 (資料顯示到前台)

- API 文件(getRequestDetail)

```
const getRequestDetail = async (modelRequestID) => {
    const output = await axios.get(DX_API.REQUEST_DETAIL.replace(':modelRequestID', modelRequestID))
    .then(response => {
        if (response.status !== StatusCodes.OK) {
            throw new Error(response.data);
        }
        return response.data;
    })
    .then(data => {
        return data.data;
    })
    .catch(error => {
        console.log(error)

        return {};
    })

return output;
}
```

```
        <br/>
```

- 主程式

```
<template>
    <div class="md:container md:mx-auto px-32 py-10">
        <div class="">
            <button
                type="button"
                @click="back"
                class="py-3 px-4 inline-flex justify-center items-center gap-2 rounded-md border border-transparent font-semibold text-gray-500
hover:text-gray-700 focus:outline-none focus:ring-2 ring-offset-white focus:ring-gray-500 focus:ring-offset-2 transition-all text-sm"
            >
                {{ '< Model Request' }}
            </button>
        </div>
        <div>
            <p class="pt-10 font-semibold text-3xl text-gray-800">
                {{ requestData.requestName }}
            </p>
            <div class="flex items-center pt-2">
                <p class="font-medium text-sm text-gray-800">by</p>
                <p class="font-medium text-gray-500 text-sm px-2">
                    {{ requestData.name }}
                </p>
            </div>
            <div class="pt-2">
                <p class="font-medium text-gray-800 text-sm">
                    {{ requestData.description }}
                </p>
            </div>
            <div class="relative flex py-12 items-center">
                <div class="flex-grow border-t-2 border-gray-200"></div>
            </div>
        </div>
    </div>
</template>
<script setup>
import { ref, onMounted } from 'vue';
import { useRouter } from 'vue-router';

import { getRequestDetail } from '../utils/dx-api.js';

const router = useRouter();

const requestData = ref({
    requestName: '',
    name: '',
    description: '',
    industry: '',
    problem: '',
    keyFunctionalities: '',
    inoutDescription: '',
    excepting: '',
});

const back = () => {
    router.push('/test');
};

onMounted(async () => {
    const modelRequestID = router.currentRoute.value.params.modelRequestID;
    const requestDetail = await getRequestDetail(modelRequestID);
    requestData.value = requestDetail;
```

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

```
<br/>
```

Vue Router

```
- Paper resource

    [https://book.vue.tw/CH4/4-2-route-settings.html](https://book.vue.tw/CH4/4-2-route-settings.html)

    [https://router.vuejs.org/guide/essentials/nested-routes.html](https://router.vuejs.org/guide/essentials/nested-routes.html)

    [https://hackmd.io/@FortesHuang/SyMAtS6qH](https://hackmd.io/@FortesHuang/SyMAtS6qH)

    [https://book.vue.tw/CH4/4-3-router-link.html](https://book.vue.tw/CH4/4-3-router-link.html)

<br/>

<br/>
```

MVC

```
 [https://www.geeksforgeeks.org/difference-between-mvc-mvp-and-mvvm-architecture-pattern-in-android/](https://www.geeksforgeeks.org/difference-between-mvc-mvp-and-mvvm-architecture-pattern-in-android/)

## 模型—視圖—控制器(MVC) 模式

MVC模式建議將代碼分為3個組件。在創建應用程序的類/文件時，開發人員必須將其分類為以下三個層之一:

- **模型(Model):** 該組件存儲應用程序數據。它對界面沒有任何了解。模型負責處理領域邏輯(真實業務規則)並與數據庫和網絡層通信。

- **視圖(View):** 它是UI(用戶界面)層，包含在屏幕上可見的組件。此外，它提供了存儲在模型中的數據的可視化，並為用戶提供交互。

- **控制器(Controller):** 此組件建立視圖和模型之間的關係。它包含核心應用程序邏輯，並得知用戶的響應並根據需要更新模型。

[]()

# MVC、MVP 和 MVVM 設計模式之間的區別

 | **MVC（模型視圖控制器）** | **MVP（模型視圖演示者）** | **MVVM（模型視圖視圖模型）** |

| ---- | ---- | ---- | | 最古老的軟件架構之一 | 作為軟件架構的第二次迭代而開發，是 MVC 的進步。 | 行業認可的應用程序架構模式。 | | UI（**視圖**）和數據訪問機制（**模型**）緊密耦合。 | 它通過使用**Presenter作為Model**和**View**之間的通信通道，解決了**View**依賴的問題。 | 這種架構模式更加事件驅動，因為它使用數據綁定，因此可以輕鬆地將**核心業務邏輯**與**視圖**分離。 | | **控制器**和**視圖**以一對多的關係存在。一個控制器可以根據需要的操作選擇不同的視圖。 | **Presenter**和**View**之間存在一對一的關係，因為一個 Presenter 類一次管理一個 View。 | 多個**View**可以映射到一個**ViewModel**，因此View和ViewModel之間存在一對多的關係。 | | 視圖不了解**控制器**。_ | View引用了**Presenter** 。 | View引用**了ViewModel** | | 由於代碼層緊密耦合，因此很難進行更改和修改應用程序功能。 | 代碼層是鬆散耦合的，因此很容易在應用程序代碼中進行修改/更改。 | 易於在應用程序中進行更改。但是，如果**數據綁定邏輯**太複雜，則調試應用程序會有點困難。 | | **用戶輸入由控制器**處理。 | 視圖是應用**程序**的入口點 | 視圖接受用戶的輸入並充當應用程序的入口點。 | | 僅適用於小型項目。 | 非常適合簡單和復雜的應用。 | 不適合小型項目。 | | **對單元測試的**支持有限。 | **單元測試**很容易進行，但 View 和 Presenter 的緊密結合可能會使其變得有些困難。 | **該架構中的單元**可測試性最高。 | | 該架構對Android API具有高度依賴性。 | 它對 Android API 的依賴性較低。 | 對 Android API 的依賴性較低或沒有。 | | 它不遵循模塊化和單一責任原則。 | 遵循模塊化和單一責任原則。 | 遵循模塊化和單一責任原則。 |

# i18n 建立

```
- 基礎建設建立

    <br/>

- 使用方法

    - <template> 內使用
```

```
{{ $t('中文內容') }}
```

```
    - <script> 內使用

        由於當語言切換後，只會直接重新變更 template 內的資訊，script 不會影響，因此需要監測語言變數，來主動更新數據
```

```
  // 先使用 watch 監測語言變數
  watch: {
    lang(newLang) {
      console.log('Current language:', newLang);
```
```

```
      this.RefreshData(); // 根據需要調用方法
    },
  // 變數依賴更新
  computed: {
    lang() {
      return this.$store.getters.getGlobalUILang;
    },
  },

  // 其他觸發更新的 function 依實際情境參考
  methods: {
      async RefreshData() {
              ...
        var netPosition = JSON.parse(JSON.stringify(lineTemplete));
        netPosition.label = this.$t('淨持倉');
        ...
    }
  },
```

- `computed` 是 Vue 中的一個特殊屬性，`computed` 之所以能做到自動更新，是因為 Vue 的響應式系統會追蹤它依賴的數據。當數據發生變化時，Vue 會自動通知依賴這些數據的 `computed` 屬性進行重新計算。這裡是如何運作的:

1. **依賴收集**:當 `computed` 屬性第一次計算時，Vue 會記錄它所依賴的數據。

1. **依賴追蹤**:當依賴的數據變化時，Vue 的響應式系統會通知這些變化，並且觸發 `computed` 屬性的重新計算。

