

Vue with vue-cli 教學

Vue with vue-cli 教學

環境安裝

安裝 node.js

安裝 vue-cli

安裝 Visual Studio Code

建立 Vue 專案

Vue 生命週期

~ mounted

mounted ~ vm.\$destroy()

vm.\$destroy() ~ destroyed

階層化的 Vue 生命週期

Vue App Instance

Components

component 物件屬性

Data-binding

ViewModel to View

View to ViewModel

陣列渲染 v-for

條件渲染 v-if、v-else-if、v-else、v-show

陣列渲染與條件渲染一起使用

開始實作 vue 專案

建立 Vue 實體

建立商品列表頁

使用 filters 來將數值轉換成金額顯示

建立商品詳情內容

輸入欲購買商品數量

利用 Router 切換頁面內容

利用 service 處理資料服務

Unit test

ProductApp.vue

ProductList.vue

ProductInfo.vue

Currency filter

附錄

eslint

Computed Caching vs Methods

Copmuted vs Watcher

環境安裝

安裝 node.js

由於前端開發時使用的套件都需要透過 `npm` 套件管理工具來安裝，以及 `vue-cli` 皆使用 `node.js` 來進行，因此需要先安裝 `node.js`

網址：<https://nodejs.org/en/>

安裝 LTS 版本即可

安裝 vue-cli

vue-cli 具備能更快速的新建專案、components 等內容，以及加速開發等優點（如更新項目立即反映到瀏覽器），因此安裝 vue-cli 可以視為實作 vue.js 專案首要的工具

在 command 輸入以下指令來全域安裝 vue-cli

```
npm install -g @vue/cli
```

npm install 為安裝套件的指令，-g 表示全域安裝

安裝 Visual Studio Code

以 Visual Studio Code (VS Code) 作為 Vue 專案開發的編輯工具，因其具有多樣的開源工具支援，以及能夠在編輯器內使用 command line，因此推薦使用它來開發

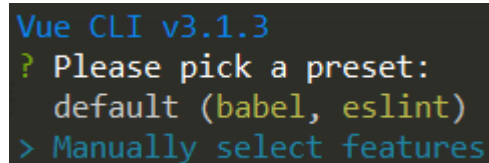
網址：<https://code.visualstudio.com/Download>

建立 Vue 專案

利用 vue-cli 來建立新的專案

```
vue create vue-tutorial
```

手動加入需要的套件



```
Vue CLI v3.1.3
? Please pick a preset:
  default (babel, eslint)
> Manually select features
```

分別選擇下列套件安裝

- Babel (ES 語法轉換)
- Router (Router 切換)
- Vuex (狀態管理)
- Linter / Formatter (code style)
- Unit Testing
- E2E Testing

```
? Check the features needed for your project:
(*) Babel
( ) TypeScript
( ) Progressive Web App (PWA) Support
(*) Router
(*) Vuex
( ) CSS Pre-processors
(*) Linter / Formatter
(*) Unit Testing
>(*) E2E Testing
```

安裝 history mode for router，輸入 Y，

```
Vue CLI v3.1.3
? Please pick a preset: Manually select features
? Check the features needed for your project: Babel, Router, Vuex, Linter, Unit, E2E
? Use history mode for router? (Requires proper server setup for index fallback in production) (Y/n)
```

Code Style 選擇 ESLint + Airbnb config，並設定儲存時檢查 Lint on save

```
? Pick a linter / formatter config:
  ESLint with error prevention only
> ESLint + Airbnb config
  ESLint + Standard config
  ESLint + Prettier
```

```
? Pick a linter / formatter config: Airbnb
? Pick additional lint features: (Press <space> to select, <a> to toggle all, <i> to invert selection)
>(*) Lint on save SCRM_V2
( ) Lint and fix on commit
```

單元測試框架使用 Jest

```
? Pick a linter / formatter config: Airbnb
? Pick additional lint features:
? Pick a unit testing solution:
  Mocha + Chai
> Jest
```

E2E 測試框架使用 Nightwatch

```
? Pick a E2E testing solution:
  Cypress (Chrome only)
> Nightwatch (Selenium-based)
```

將設定值另外存放，選擇 In dedicated config files

```
? Pick a E2E testing solution: Nightwatch
? Where do you prefer placing config for Babel, PostCSS, ESLint, etc.? (Use arrow keys)
> In dedicated config files
  In package.json
```

是否將設定值作為預設設定，這裡可自由設置

完整設置內容：

```
Vue CLI v3.1.3
? Please pick a preset: Manually select features
? Check the features needed for your project: Babel, Router, Vuex, Linter, Unit, E2E
? Use history mode for router? (Requires proper server setup for index fallback in production) Yes
? Pick a linter / formatter config: Airbnb
? Pick additional lint features: Lint on save
? Pick a unit testing solution: Jest
? Pick a E2E testing solution: Nightwatch
? Where do you prefer placing config for Babel, PostCSS, ESLint, etc.? In dedicated config files
? Save this as a preset for future projects? (y/N) |
```

安裝完成後，依照指示輸入

```
cd vue-tutorial
npm run serve
```

頁面在 localhost:8081 監聽，利用網頁瀏覽器打開即可看到成功建置畫面

[Home](#) | [About](#)



Welcome to Your Vue.js App

For a guide and recipes on how to configure / customize this project,
check out the [vue-cli documentation](#).

Installed CLI Plugins

[babel](#) [eslint](#) [unit-jest](#) [e2e-nightwatch](#)

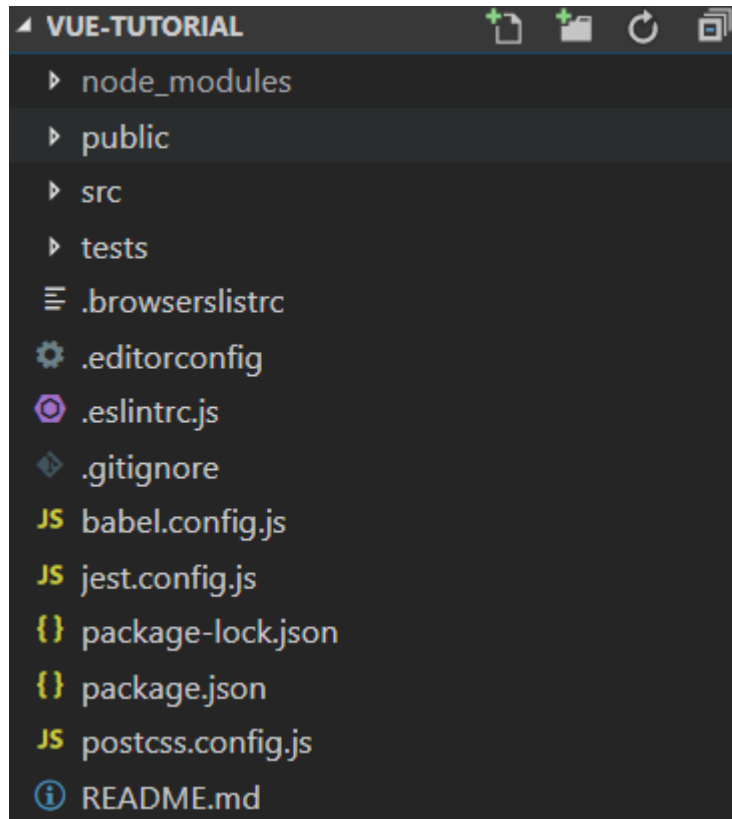
Essential Links

[Core Docs](#) [Forum](#) [Community Chat](#) [Twitter](#) [News](#)

Ecosystem

[vue-router](#) [vuex](#) [vue-devtools](#) [vue-loader](#) [awesome-vue](#)

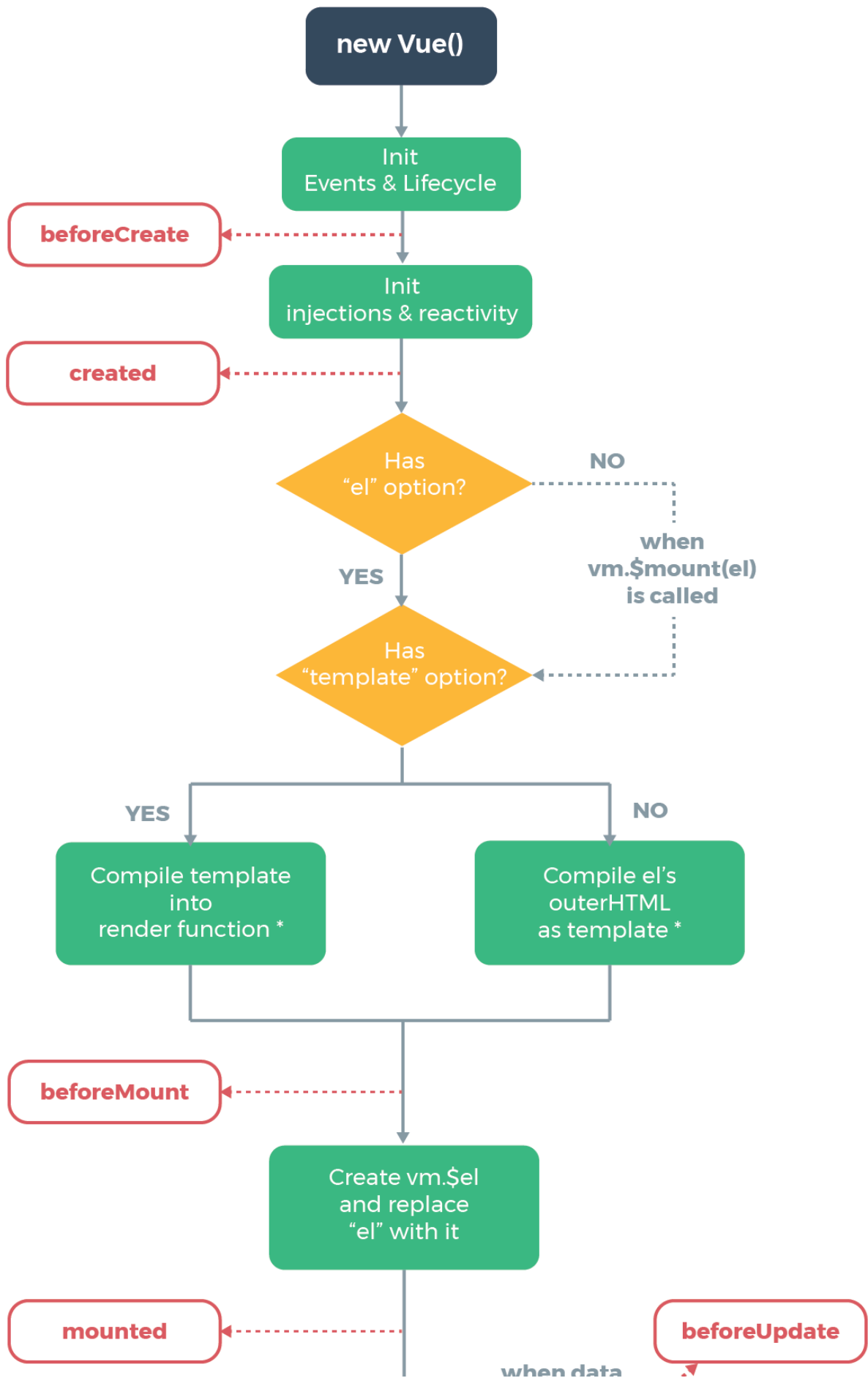
利用 VS Code 打開專案可以看到如下架構

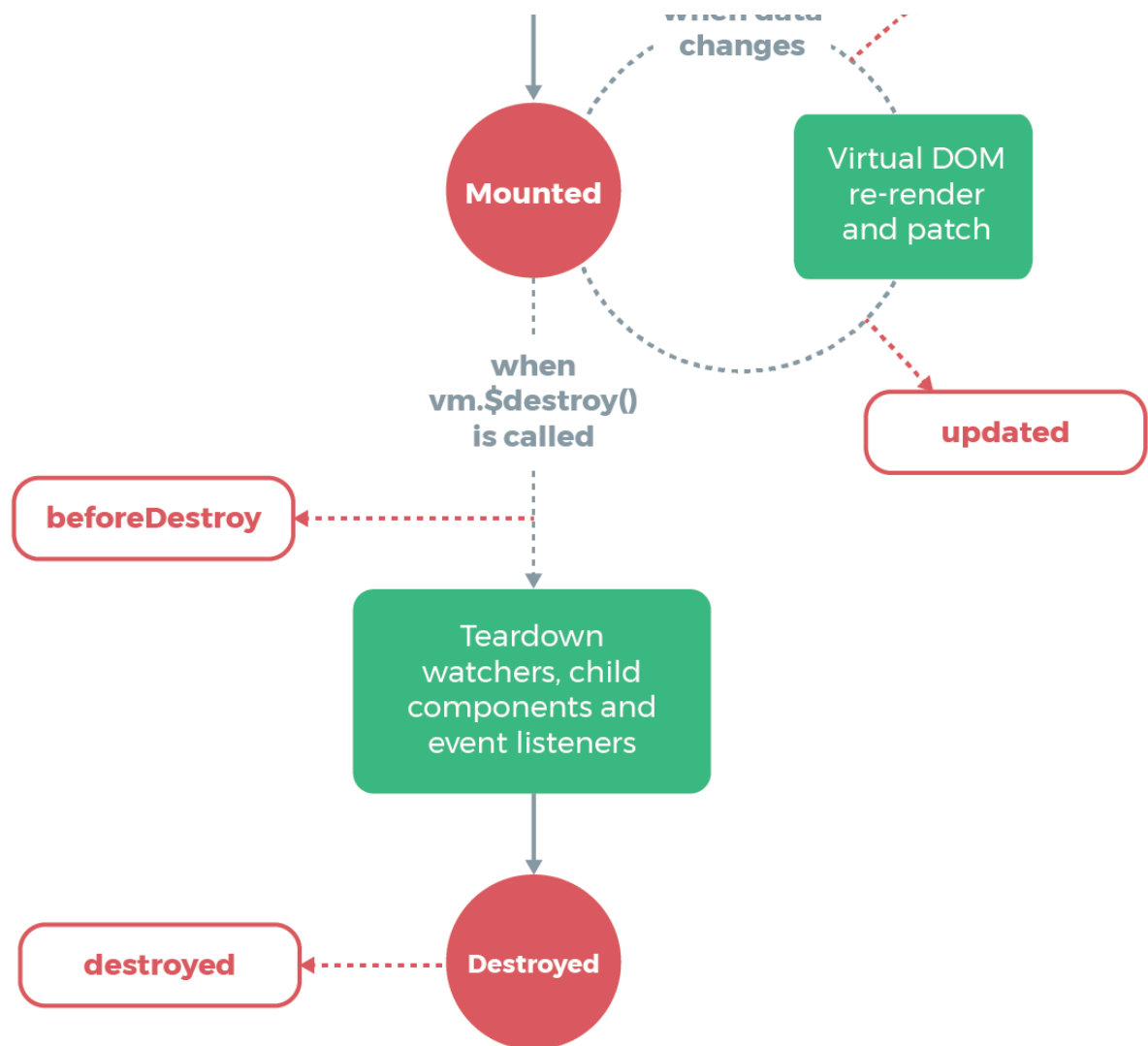


- node_modules：存放相關套件的位置
- public：以 SPA 方式設計的網頁，作為外框所需資源存放的位置
- src：vue 核心開發內容存放的位置
 - assets：圖片等靜態檔案存放位置
 - components：components 存放位置
 - views：views 存放位置
 - main.js：Vue app 載入設置及實際載入 Vue instance 的 script
 - router.js：安裝了 Router 自動產生的 script，在此設定 router config
 - store.js：安裝了 Vuex 自動產生的 script，在此控制 store 狀態
- tests：與測試相關的設置
- 其他相關設置：待使用時一一介紹

Vue 生命週期

首先先從 Vue 的生命週期開始了解





* template compilation is performed ahead-of-time if using a build step, e.g. single-file components

如上圖，先將生命週期以 mounted / beforeDestroy 作為分隔點切成三個部分，先從 mounted 前開始

~ mounted

任何 Vue App 的 Instance 皆是由 new Vue() 開始建立，並可在傳入的 options object 定義 Instance Lifecycle Hooks 來自訂各週期呼叫的方法 (圖中白底紅字的部分)

1. 首先，初始化 Vue lifecycle 及事件方法
2. 呼叫 Instance Lifecycle Hooks 的 `beforeCreate` 方法，須注意此時 `$data` 還無法調用
3. 初始化注入相依內容及資料綁定
4. 呼叫 Instance Lifecycle Hooks 的 `created` 方法，此時 Instance 已初始化完成，事件與屬性已綁定完成，`$data` 已可取得，但尚未掛載 el 且尚未生成 DOM
5. 判斷 options 中是否有 el 屬性，若有則繼續進行下一步，若無則等待 `.$mount(el)` 被呼叫

- 判斷 options 中是否有 template 屬性，若有，編譯之並放入到 render function 中，若無則以 el 的 outerHTML 來編譯

這裡會判斷是否有 render 屬性，若有則優先以 render 來 compile，優先序為 render -> template -> el 的內容，render 回傳的內容為 VNode (Virtual Node) 物件，並不是實際的 DOM 物件

- 呼叫 Instance Lifecycle Hooks 的 `beforeMount` 方法，此階段尚未掛載 render 後的內容，尚未將 el 內容取代
- 將 render 調用後的內容取代 el
- 呼叫 Instance Lifecycle Hooks 的 `mounted` 方法，此階段已經將 render 的內容取代 el，且 DOM 已經綁定完成，可以在此階段發送 ajax 來更新頁面資料

可以透過下列程式碼來一一檢驗，須注意：若是使用 **vue-cli** 來建置專案，預設為 **pre-build** 模式，因此在 **main.js** 內放入 **template** 屬性會出現如下的錯誤

```
► [Vue warn]: You are using the runtime-only build of Vue where the template compiler is not available. Either pre-compile the templates into render functions, or use the compiler-included build.
vue.runtime.esm.js?2b0e:587
(found in <Root>)
```

在根目錄加入 **vue.config.js** 檔案，並設置 **runtimeCompiler** 為 **true**，以在執行期編譯 **template**

```
/* vue.config.js */
module.exports = {
  runtimeCompiler: true
};
```

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <link rel="icon" href="<%= BASE_URL %>favicon.ico">
    <title>vue-tutorial</title>
  </head>
  <body>
    <noscript>
      <strong>we're sorry but vue-tutorial doesn't work properly without JavaScript
enabled. Please enable it to continue.</strong>
    </noscript>
    <div id="app">
      <div>{{message}}</div>
    </div>
    <div id="app1">
      <div>{{message}}</div>
    </div>
    <div id="app2">
      <div>{{message}}</div>
    </div>
    <div id="app3">
```

```
    <div>{{message}}</div>
  </div>
  <!-- built files will be auto injected -->
</body>
</html>
```

```
/* main.js */
import InitApp from './InitApp'

// no el instance
var vm = new Vue({
  data: { message: 'call mounted to create instance' },
  beforeCreate: function() {
    console.log(this.message);
  },
  created: function() {
    console.log(this.message);
  },
  beforeMount: function() {
    console.log(document.getElementById('app'));
  },
  mounted: function() {
    console.log(document.getElementById('app'));
  }
});

// uncomment to check if template mounted
// vm.$mount('#app')

// no template instance
new Vue({
  el: '#app1',
  data: { message: 'no template in options' },
  beforeCreate: function() {
    console.log(this.message);
  },
  created: function() {
    console.log(this.message);
  },
  beforeMount: function() {
    console.log(document.getElementById('app1'));
  },
  mounted: function() {
    console.log(document.getElementById('app1'));
  }
});

// with template instance
new Vue({
  el: '#app2',
  data: { message: 'with template in options' },
  beforeCreate: function() {
```

```

        console.log(this.message);
      },
      created: function() {
        console.log(this.message);
      },
      beforeMount: function() {
        console.log(document.getElementById('app2'));
      },
      mounted: function() {
        console.log(document.getElementById('app2'));
      }
    }
  });

  // with template rendered by render function instance
  new Vue({
    el: '#app3',
    beforeCreate: function() {
      console.log(this.message);
    },
    created: function() {
      console.log(this.message);
    },
    beforeMount: function() {
      console.log(document.getElementById('app3'));
    },
    mounted: function() {
      console.log(document.getElementById('app3'));
    },
    render: h => h(InitApp);
  });

```

```

<!-- InitApp.vue -->
<template>
  <div>this template is rendered by render function</div>
</template>

```

透過頁面內容及 console 內容來確認每個時期的輸出內容

mounted ~ vm.\$destroy()

接著我們來看 mounted 之後的生命週期，當 mounted 完成後即開始監聽資料變化，若資料有變化開始進行，須注意：**beforeUpdate** 與 **updated** 方法只有在 **mounted** 之後有資料異動才會呼叫，因此 **data** 的初始設定值在 **created** 時已經綁定，並不會呼叫 **beforeUpdate** 與 **updated**

1. 呼叫 Instance Lifecycle Hooks 的 **beforeUpdate** 方法，此時還未將 DOM 重新渲染更新
2. 重新渲染 DOM 並更新到結點上
3. 呼叫 Instance Lifecycle Hooks 的 **updated** 方法，此時資料及DOM已經更新

可以透過以下程式來觀察驗證

```

/* main.js */
new Vue({

```

```

    el: '#app',
    data: { message: 'message' },
    template: '<div><input v-model="message" /><div><span>the value is: </span><span id="message">{{ message }}</span></div></div>',
    beforeCreate: function() {
      console.log(this.message);
    },
    created: function() {
      console.log(this.message);
    },
    beforeMount: function() {
      console.log(document.getElementById('app'));
    },
    mounted: function() {
      console.log(document.getElementById('app'));
    },
    beforeUpdate: function() {
      console.log(document.getElementById('message').textContent);
    },
    updated: function() {
      console.log(document.getElementById('message').textContent);
    }
  });

```

觀察 console 輸出內容來驗證結果

vm.\$destroy() ~ destroyed

當 Instance 準備被銷毀時會進行剩下的銷毀週期

1. 呼叫 Instance Lifecycle Hooks 的 `beforeDestroy` 方法，此時還沒實際銷毀 instance
2. 銷毀 instance
3. 呼叫 Instance Lifecycle Hooks 的 `destroyed` 方法，此時 instance 已經被銷毀

可以透過以下程式來觀察驗證，增加一個按鈕來模擬觸發 `.$destroy()` 方法

```

new Vue({
  el: '#app',
  data: { message: 'message' },
  template: '<div><input v-model="message" /><div><span>the value is: </span><span id="message">{{ message }}</span></div><button @click="onDestroy">Self destruct!</button></div>',
  beforeCreate: function() {
    console.log(this.message);
  },
  created: function() {
    console.log(this.message);
  },
  beforeMount: function() {
    console.log(document.getElementById('app'));
  },
  mounted: function() {
    console.log(document.getElementById('app'));
  }
});

```

```

},
beforeUpdate: function() {
  console.log(document.getElementById('message').textContent);
},
updated: function() {
  console.log(document.getElementById('message').textContent);
},
beforeDestroy: function() {
  console.log('instance is going to be destroy...');
},
destroyed: function() {
  console.log('instance has been destroyed');
},
methods: {
  // onDestroy event
  onDestroy: function() {
    this.$destroy();
  }
}
});

```

透過按下 **Self destruct!** 來觀察 console 輸出，並在之後嘗試更改 input 值來驗證 instance 是否被銷毀

`.$destroy()` 觸發的時機為內容被移除介面時，如換頁或是透過 `v-if` 操作來附加或移除 components 等

階層化的 Vue 生命週期

每個 components 或是 views 皆有獨立完整的生命週期，最外層 instance 的掛載 (mount) 會等到內部使用到的所有 instance 皆掛載完成 (mounted) 後才會接著掛載完成，銷毀的順序亦然，觀察下列程式碼的 console 輸出來驗證結果

```

Vue.component('example', {
  template: '<div>example line</div>', beforeCreate: function() {
    console.log('example before create');
  },
  created: function() {
    console.log('example created');
  },
  beforeMount: function() {
    console.log('example before mount');
  },
  mounted: function() {
    console.log('example mounted');
  },
  beforeDestroy: function() {
    console.log('example before destroy');
  },
  destroyed: function() {
    console.log('example destroyed');
  }
});

```

```

new Vue({
  el: '#app',
  data: { message: 'message' },
  template: '<div><input v-model="message" /><div><span>the value is: </span><span id="message">{{ message }}</span></div><example></example><button @click="onDestroy">Self destruct!</button></div>',
  beforeCreate: function() {
    console.log(this.message);
  },
  created: function() {
    console.log(this.message);
  },
  beforeMount: function() {
    console.log(document.getElementById('app'));
  },
  mounted: function() {
    console.log(document.getElementById('app'));
  },
  beforeUpdate: function() {
    console.log(document.getElementById('message').textContent);
  },
  updated: function() {
    console.log(document.getElementById('message').textContent);
  },
  beforeDestroy: function() {
    console.log('instance is going to be destroy...');
  },
  destroyed: function() {
    console.log('instance has been destroyed');
  },
  methods: {
    // onDestroy event
    onDestroy: function() {
      this.$destroy();
    }
  }
});

```

Vue App Instance

透過 new Vue() 來建立新的 Vue App Instance

```

new vue({
  render: h => h(App)
}).$mount('#app')

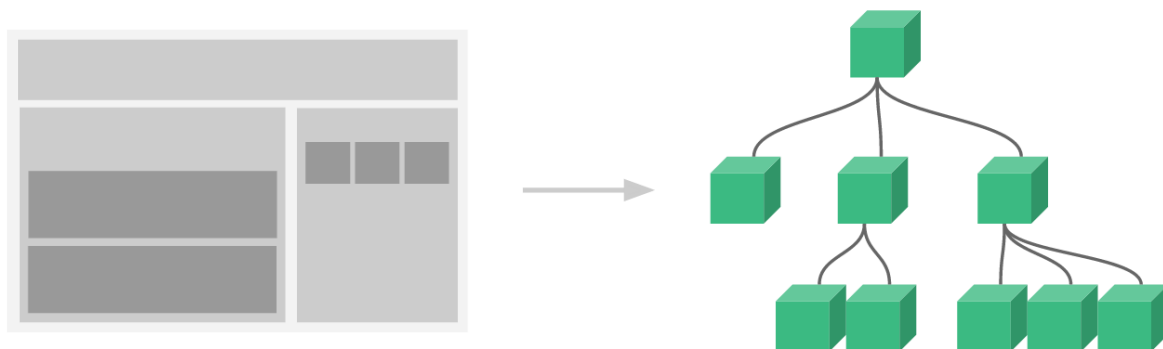
```

\$mount() 會將該實體掛載，開始進行 Vue 的生命週期，在傳入 new Vue() 的 options 參數中定義 el 屬性也能達到一樣的作用，透過 \$mount() 能自行決定掛載的時機點，注意：\$mount() 傳入的參數字串 #app 與欲取代的 DOM id 一致

Components

我們已經建立一個 Vue app instance，事實上，也可以用多個 Vue app instance 來建立功能 (如前面測試生命週期建立的多個 instance)，因此包裹最外層的 instance 可以視為一個完整的 Vue app

在 Vue 中，一個完整的頁面可以被切分為多個元件 (components)，如下圖所示



透過如樹狀結構般的 components 來組成頁面的完整功能，藉此來達到重複利用 component 的特性

另外，使用 vue-cli 建立的目錄中，可以看到 src/views 及 src/components 兩個目錄，存放在兩者中的檔案皆為 components 並無差異，唯一的差別在於其目的：會定義為 view 的 component 用來作為隨著 router 路由切換的頁面內容

一個 component 可以如下方式來建立

```
/* main.js */
vue.component('welcome', {
  template: '<h2>welcome!</h2>',
});
```

第一個參數為 component 的名稱，第二個參數為 component 物件

實際使用可以如下定義：

首先，在 index.html 新增一個 App 容器，並在此容器使用剛剛定義的 compoent 標籤 <welcome></welcome>

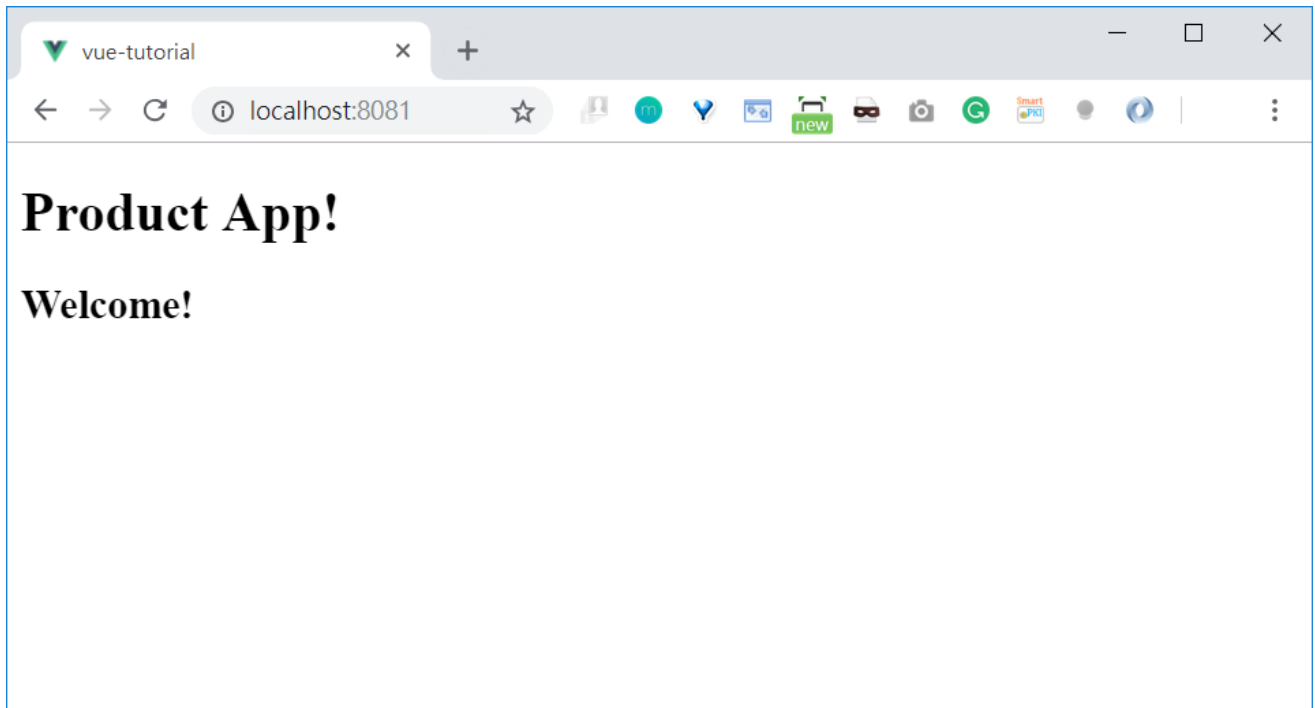
```
<!-- index.html -->
<div id="app2">
  <welcome></welcome>
</div>
```

接著在 main.js 建立一個新的 Vue app instance，加上剛剛宣告的 compoent，內容如下

```
Vue.component('welcome', {
  template: '<h2>welcome!</h2>',
});

new Vue({
  el: '#app2',
});
```

將變更的檔案儲存，可以看到新的畫面如下，Welcome! 標題已成功加入至頁面內容中



component 物件屬性

一個 component 物件常用的屬性有：

- data：作為頁面資料存放的位置，在 components 中必須為一個回傳物件的方法

```
vue.component('data-demo', {
  data: function() {
    return {
      message: 'welcome!'
    }
  }
});
```

- props：作為傳入 component 的參數

簡單的定義方式


```
Vue.component('param-demo', {
  props: ['message', 'size']
});
```

```
<param-demo message="welcome" size="10"></param-demo>
```

進階的定義方式

```
Vue.component('example', {
  props: {
    // basic type check (`null` means accept any type)
    onSomeEvent: Function,
    // check presence
    requiredProp: {
      type: String,
      required: true
    },
    // with default value
    propWithDefault: {
      type: Number,
      default: 100
    },
    // object/array defaults should be returned from a
    // factory function
    propWithObjectDefault: {
      type: Object,
      default: function () {
        return { msg: 'hello' }
      }
    },
    // a two-way prop. will throw warning if binding type
    // does not match.
    twoWayProp: {
      twoWay: true
    },
    // custom validator function
    greaterThanTen: {
      validator: function (value) {
        return value > 10
      }
    }
  }
});
```

- methods : component 中使用到的頁面方法

```
Vue.component('param-demo', {
  methods: {
    onTabClick: function() {
      ...
    }
  }
});
```

- `computed`：經過計算更新值的屬性，類似 `getter` 與 `setter`

```
new Vue({
  data: { a: 1 },
  computed: {
    // get only, just need a function
    aDouble: function () {
      return this.a * 2
    },
    // both get and set
    aPlus: {
      get: function () {
        return this.a + 1
      },
      set: function (v) {
        this.a = v - 1
      }
    }
  }
});
```

注意：在 `computed function` 中的 `this` 皆是指向 `data` 物件

[Computed Caching vs Methods](#)

[Computed vs Watcher](#)

- `template`：頁面 `html` 內容
- 其他 `Instance Lifecycle Hooks`

Data-binding

ViewModel to View

在 `Vue App` 裡，透過使用雙大括號 `{{}}` (`Mustache`) 進行資料綁定，當 `Vue` 在 `mount` 階段時，即會將 `{{}}` 取代為 `data` 中設定的資料內容，如下

```
<div>{{ message }}</div>
```

若要針對 `attribute` 來綁定，如下

```
<div v-bind:class="dynamicClass">text</div>
<div v-bind:title="message">this is message</div>
```

當 data 中的屬性值變更時，頁面上對應的資料也會即時重新渲染

注意：若在 mount 階段沒有綁定的 data 屬性，之後更改這些屬性值，頁面也不會即時重新渲染，因此要在 data 定義時就要加入屬性，若剛開始沒有值可以指定，可以用給定初始值的方式賦值

```
new Vue('example', {
  data: function() {
    return {
      message: '',
      size: 0
    }
  }
});
```

View to ViewModel

當頁面上的內容變更，例如 input 欄位經使用者變更值的情況下，可以利用 View to ViewModel 的語法來變更 ViewModel 的資料值，如下

```
<input v-model="searchText">
```

上面的內容與下相同

```
<input v-bind:value="searchText"
  v-on:input="searchText = $event.target.value">
```

實際上是透過 事件 來更新 data 屬性值

陣列渲染 v-for

使用 v-for，可以將陣列資料列舉渲染

```
<ul>
  <li v-for="item in data" :key="item.id">{{ item.name }}</li>
</ul>
```

條件渲染 v-if、v-else-if、v-else、v-show

使用 v-if，可以依據條件決定是否渲染，達到動態顯示/隱藏內容的目的

```
<div>
  <h2 v-if="condition">條件1</h2>
  <h2 v-else>條件2</h2>
</div>
```

注意：若要使用 `v-else-if`、`v-else`，需要皆在 `v-if` 之後，且需要與 `v-if` 同階層，若無，則視為無效內容

`v-show` 與 `v-if` 類似，皆可以透過條件來顯示/隱藏內容，差別在於：`v-show` 不管條件判斷如何皆會渲染，但會透過 `css` 的方式隱藏內容，`v-if` 若條件判斷為 `false`，則不會渲染，因此 `v-show` 在頁面初始渲染時有較高的成本，但在動態條件切換時能節省較多資源，`v-if` 則反之，在動態條件切換時耗用較高的資源

陣列渲染與條件渲染一起使用

不建議將 `v-for` 與 `v-if` 在同一個 `element` 中一併使用，改以透過 `computed` 的方式，先將要顯示的內容過濾後，再使用 `v-for` 渲染

```
<!-- bad -->
<ul>
  <li v-for="item in products" v-if="item.isActive" :key="item.id">{{ item.name }}
</li>
</ul>

<!-- good -->
<ul>
  <li v-for="item in activeProducts" :key="item.id">{{ item.name }}</li>
</ul>
```

```
new Vue({
  data: {
    products: [ ... ]
  },
  computed: {
    activeProducts: function() {
      return this.products.filter(product => product.isActive);
    }
  }
});
```

若 `v-for` 與 `v-if` 一併使用，則 `v-for` 有較高的優先度

開始實作 vue 專案

在了解 `Vue` 的生命週期與 `Components` 的使用後，我們開始實作第一個 `Vue` 專案

使用 `vue-cli` 建置專案，可以看到部分基礎檔案已經存在，現在我們先忽略這些已經建立好的檔案，一步一步開始實作 `vue` 專案

建立 Vue 實體

Single Page Application (SPA) 的其中一個特色為：透過定義一個隨著網址或路由 (router) 切換而改變的容器來包覆動態內容，而不是重新載入整個頁面，以減少重複資源的重新載入

在 public 資料夾下的 index.html 即作為包覆整個頁面的外層框，可以在內容看到

```
<div id="app"></div>
```

此處的 app 僅代表 `<div>` 的 id，實際上 Vue 創建的 DOM 即會以 id 來尋找需要取代的對象，保留這份檔案的內容，更改 main.js 的內容來建立新的 vue app instance

透過 vue-cli 自動建立的檔案中，可以找到 main.js 作為 Vue 的實體建立設置，內容已經包含：

```
new Vue({
  router,
  store,
  render: h => h(App),
}).$mount('#app');
```

先將已有的內容註解，放入以下內容

```
new Vue({
  render: h => h(ProductApp),
}).$mount('#app');
```

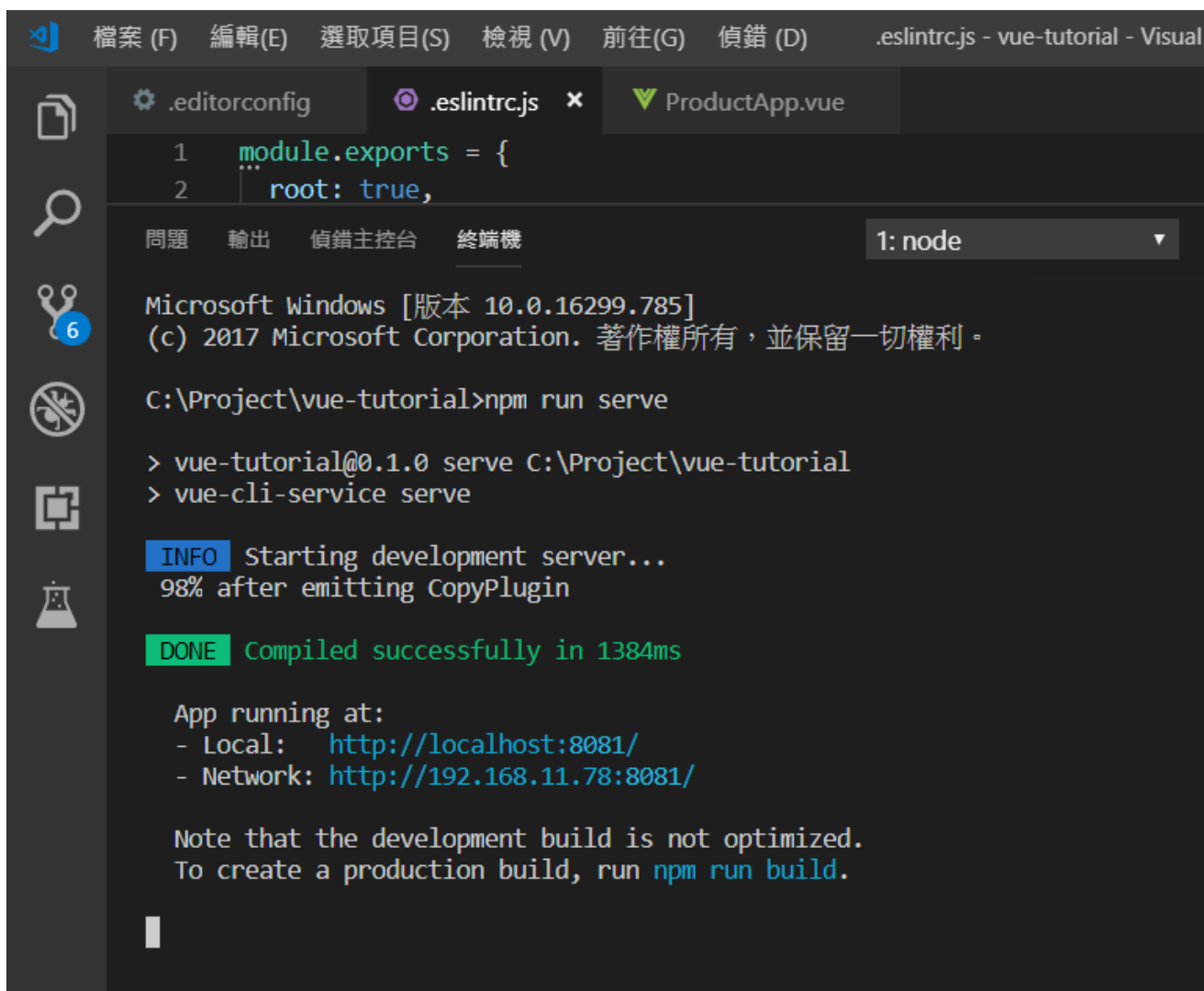
在 src 底下新增一個 ProductApp.vue 對象，並在 main.js 引入

```
<!-- ProductApp.vue -->
<template>
  <h1>Product App!</h1>
</template>
```

```
/* main.js */
import ProductApp from './ProductApp.vue';

new Vue({
  render: h => h(ProductApp),
}).$mount('#app');
```

由於使用 VS Code 來進行開發，可以使用 VS Code 提供的終端機來開啟伺服器監聽：按下 `ctrl` + ``` 來開啟終端機，並輸入 `npm run serve` 啟動伺服器監聽



The screenshot shows the Visual Studio Code interface with the terminal open. The terminal output is as follows:

```
Microsoft Windows [版本 10.0.16299.785]
(c) 2017 Microsoft Corporation. 著作權所有，並保留一切權利。

C:\Project\vue-tutorial>npm run serve

> vue-tutorial@0.1.0 serve C:\Project\vue-tutorial
> vue-cli-service serve

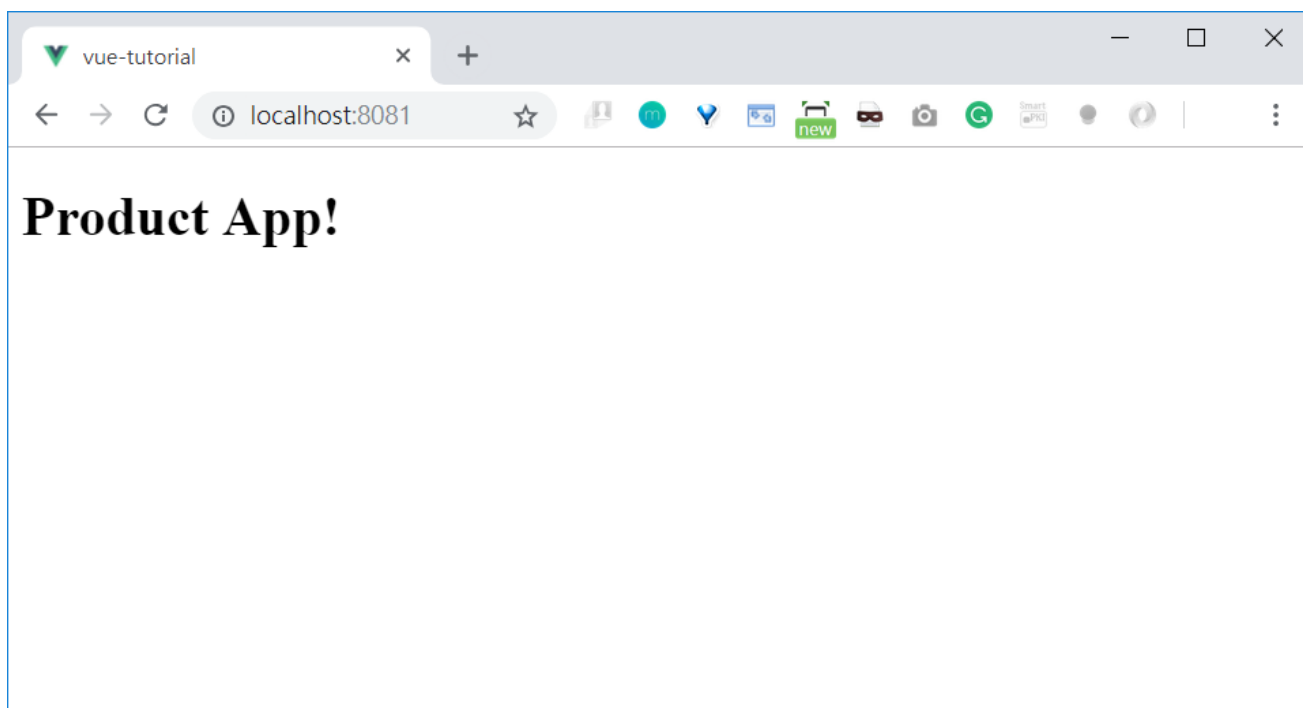
INFO Starting development server...
98% after emitting CopyPlugin

DONE Compiled successfully in 1384ms

App running at:
- Local: http://localhost:8081/
- Network: http://192.168.11.78:8081/

Note that the development build is not optimized.
To create a production build, run npm run build.
```

可以看到開啟的監聽 port 為 8081，在瀏覽器輸入就可以看到成功顯示的畫面



vue-cli-service 會自動監聽檔案變更，在被監聽的檔案有異動並按下儲存時，瀏覽器會自動更新頁面

在建置時會看見 terminal 顯示若干錯誤，此部分錯誤與 [eslint](#) 相關設置有關

建立商品列表頁

接著建立商品列表頁，商品列表頁要列出多項商品卡片，每一個商品卡片都要顯示商品名稱、簡介、縮圖、價格

有別於先前建立 components 的方式，為了達到維護簡易性，在 components 目錄下加入 ProductList.vue 來宣告新的 components，一個 .vue 檔案包含三個區塊

- template：component 的 html 內容
- style：component 的 css 樣式設置，在 component 的樣式只侷限在該 component 及其子 component 的範圍
- script：script 內容，一般會將 component 的定義類別放在此區塊內

```
<template>
  <div>
    <div class="product-container" v-for="item in products" :key="item.id">
      <div class="product-image">
        
      </div>
      <div class="product-info">
        <div class="product-title">{{ item.name }}</div>
        <div class="product-content">{{ item.desc }}</div>
        <span class="product-price">{{ item.price }}</span>
      </div>
    </div>
  </div>
</template>

<script>
export default {
  name: 'ProductList',
  data() {
    return {
      products: [
        {
          id: 1,
          name: 'Car',
          imgUrl: 'https://images.pexels.com/photos/1200458/pexels-photo-1200458.jpeg?
auto=compress&cs=tinysrgb&h=350',
          desc: 'A luxury car you won\'t miss it.',
          price: 199.99,
        },
        {
          id: 2,
          name: 'Bike',
          imgUrl: 'https://images.pexels.com/photos/1239460/pexels-photo-1239460.jpeg?
auto=compress&cs=tinysrgb&h=350',
          desc: 'A durable bike you\'ve never ride.',
          price: 25.00,
        },
      ],
    }
  },
}
```

```
    ],
  };
},
};
</script>

<style>
.product-container {
  display: flex;
  flex-direction: row;
  justify-content: flex-start;
  padding: 5px;
  margin: 4px 2px;
  background: #eee;
  border-radius: 3px;
  font-family: sans-serif;
}

.product-image {
  display: flex;
  width: 30%
}

.product-image img {
  align-self: center;
  width: 100%;
  height: auto;
}

.product-info {
  display: flex;
  padding: 5px 8px;
  flex-direction: row;
  flex-wrap: wrap;
  align-content: space-between;
  width: 70%;
}

.product-title {
  width: 100%;
  font-size: 1.5em;
  font-weight: bold;
}

.product-content {
  width: 100%;
  font-size: 1em;
}

.product-price {
  width: 100%;
  font-size: 1.1em;
  color: rgba(255, 30, 0, 0.884);
}
```



```
text-align: right;
}
</style>
```

若再 VS Code 有安裝 Vue VSCode Snippets 套件，輸入 vbase 再按下 `tab` 就能快速產生 vue 樣板
並在 ProductApp.vue 加入 component 的引用，以及使用 product-list element

```
<template>
  <div id="productApp">
    <h1>Product App!</h1>
    <product-list></product-list>
  </div>
</template>

<script>
import ProductList from '@components/ProductList';

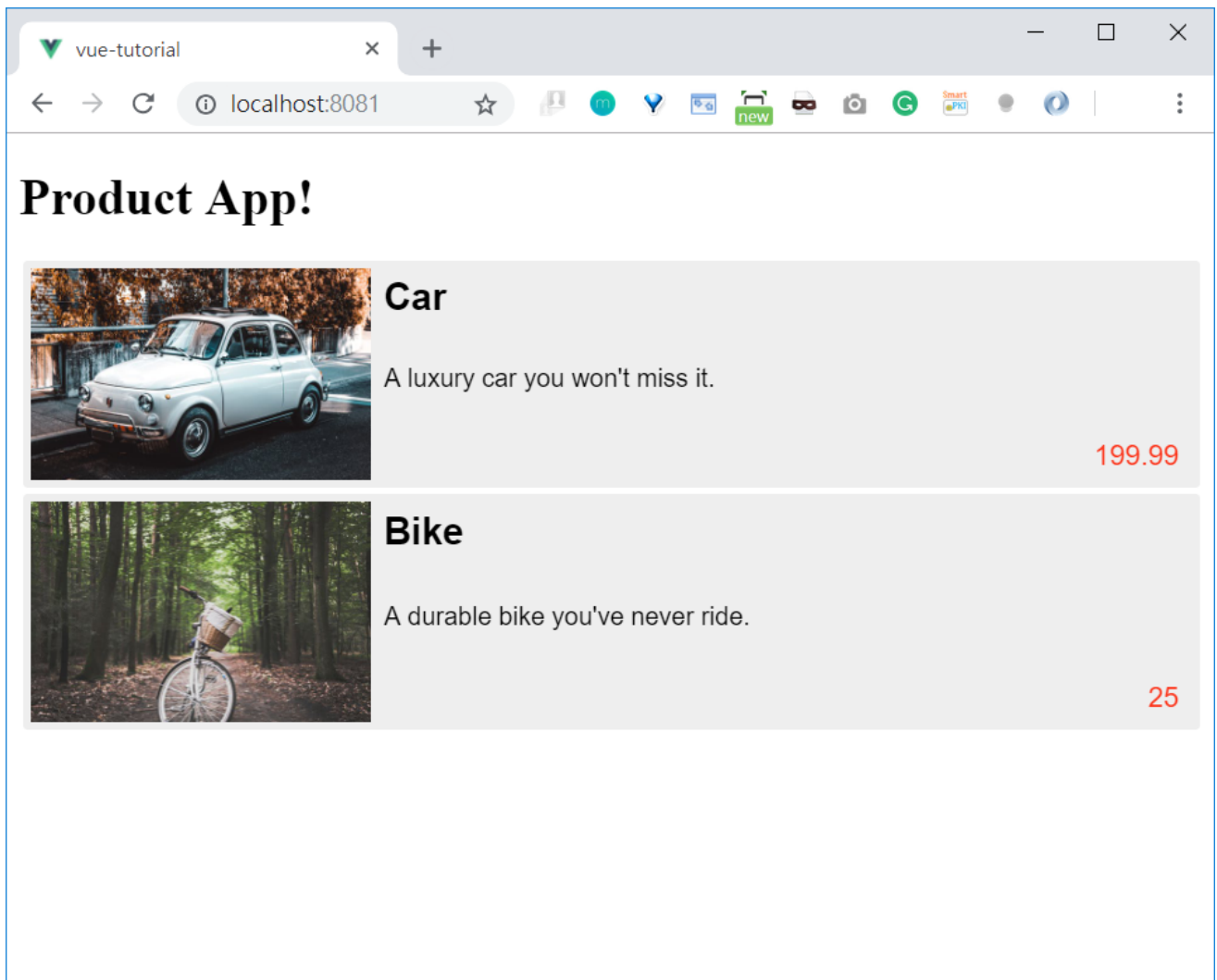
export default {
  components: {
    ProductList,
  },
};
</script>
```

有別於先前的範例，在 main.js 使用 component 的方法

```
Vue.component('param-demo', { ... });
```

使用 Vue.component 宣告的 component 範圍為全域 component，一旦在全域宣告，任何 Vue App Instance 範圍底下都可以使用該 component 不用再另外引入

儲存後畫面如下：



透過使用條件渲染 `v-for` 將多個商品資料加入至 `product-container` 中，並利用 `data-binding` 的方式將 `data` 中的商品資訊顯示在頁面

`v-bind:attribute` 可以縮寫成 `:attribute`

如此已經完成商品列表頁的功能

使用 **filters** 來將數值轉換成金額顯示

先前完成的內容，商品價格的顯示還不夠完整，應該還要在前面加入幣值符號，因考量到多處都需要加上，可以使用 `filters` 來完成，`filters` 使用的方法如下

```
<!-- in mustaches -->
{{ price | currency }}

<!-- in v-bind -->
<div v-bind:id="rawId | formatId"></div>
```

透過使用 `|` 符號 (pipe)，在後面加入 `filters` 名稱，也可以使用多個 `filters` 來處理資料

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

定義 filters 的方式如同 components，可以在單個 component 引用，或是透過 Vue.filter 註冊，這裡我們考量幣值符號顯示 filter 會在多個頁面顯示，所以我們以全域的方式註冊

先在 src 底下新增一個 filters 資料夾，並在該資料夾底下新增 Currency.js 檔案

```
export default function(value) {  
  return `${value}`;  
}
```

然後在 main.js 全域註冊，注意：必須加在宣告 Vue App Instance 前

```
import Currency from './filters/Currency';  
  
...  
vue.filter('currency', Currency);  
  
new Vue({ ... });
```

最後在需要使用到 filter 的地方使用 pipe | 符號

```
<span class="product-price">{{ item.price | currency }}</span>
```

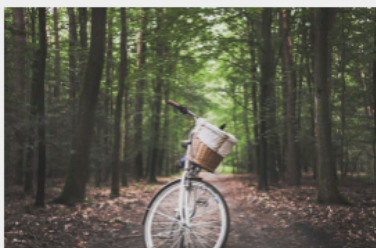
完成的內容如下



Car

A luxury car you won't miss it.

\$ 199.99



Bike

A durable bike you've never ride.

\$ 25

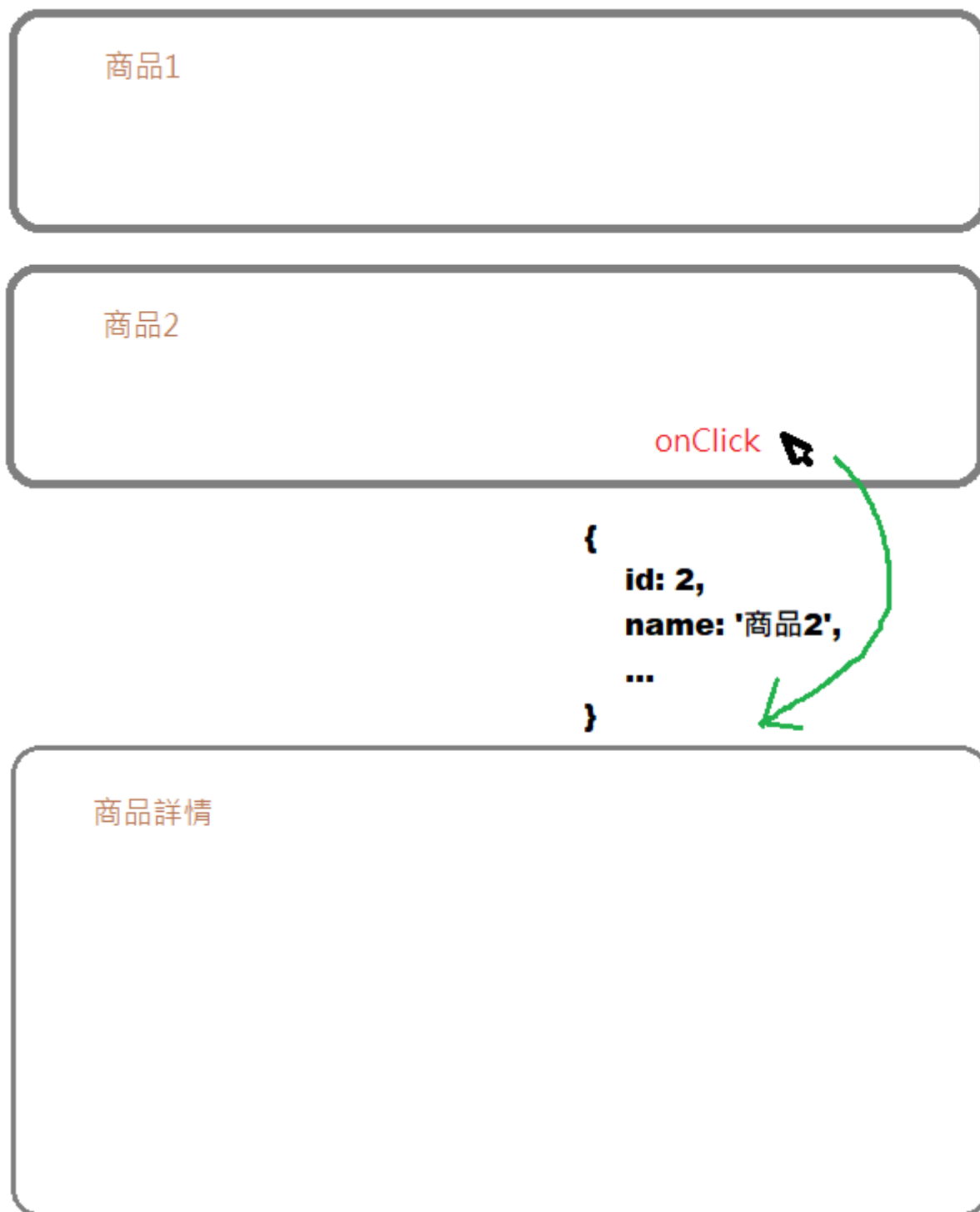
若是需要個別在 components 引用，options 使用方式如下：

```
{  
  data: function() { ... },  
  filters: {  
    currency: function(value) { ... }  
  }  
}
```

建立商品詳情內容

接著實作商品詳情內容，除了需要隨著點擊的商品顯示不同資訊

在實作功能前，先以下面圖片說明實作流程



利用 **props** 將外部資料導至 **component** 內，在商品列表每個項目的 **onClick** 事件將該商品資料動態的更改 **props** 值，如此達到動態更改商品資訊區塊內容

在 **components** 底下新增 **ProductInfo.vue** 檔案

```
<template>  
  <div>
```

```
<h2>Product Info</h2>
<div class="product-container">
  <div class="product-title">{{ product.name }}</div>
  <div class="product-image">
    
  </div>
  <div class="product-info">
    <div class="product-content">{{ product.desc }}</div>
    <span class="product-price">{{ product.price | currency }}</span>
  </div>
</div>
</div>
</template>
```

```
<script>
export default {
  name: 'ProductInfo',
  props: {
    item: {
      Object,
      default: {},
    },
  },
  data() {
    return {
      product: this.item,
    };
  },
  watch: {
    item(value) {
      this.product = value;
    },
  },
};
</script>
```

```
<style scoped>
.product-container {
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
  justify-content: flex-start;
  background: #eee;
  border-radius: 3px;
  font-family: sans-serif;
}

.product-title {
  width: 100%;
  padding: 10px 20px;
  font-size: 2.5em;
}
```

```

.product-image {
  display: flex;
  flex-direction: column;
  align-items: center;
  width: 100%;
  padding: 10px;
}

.product-image img {
  align-self: center;
  width: 60%;
  height: auto;
  border: 1px solid #ddd;
  border-radius: 5px;
}

.product-info {
  width: 100%;
  padding: 10px;
}

.product-content {
  width: 100%;
  font-size: 1.5em;
}

.product-price {
  color: rgba(255, 30, 0, 0.884);
  font-size: 2em;
  float: right;
}
</style>

```

注意之後的 style 標籤上都加上了 `scoped` attribute，目的是宣告該 style 的內容只針對該 component 的樣式設定

透過使用 props 接收外部資料，並讓 `data.product` 初始值指定為該商品資料

```

{
  ...
  props: {
    item: {
      type: Object,
      default() {
        return {};
      },
    },
  },
  data() {
    return {
      product: this.item,
    };
  },
}

```

```
...  
}
```

另外，因為商品詳細資料會隨著選擇的商品不同而有不同內容，因此在 `watch` 加入 `item` 監控，以更新 `data.product`

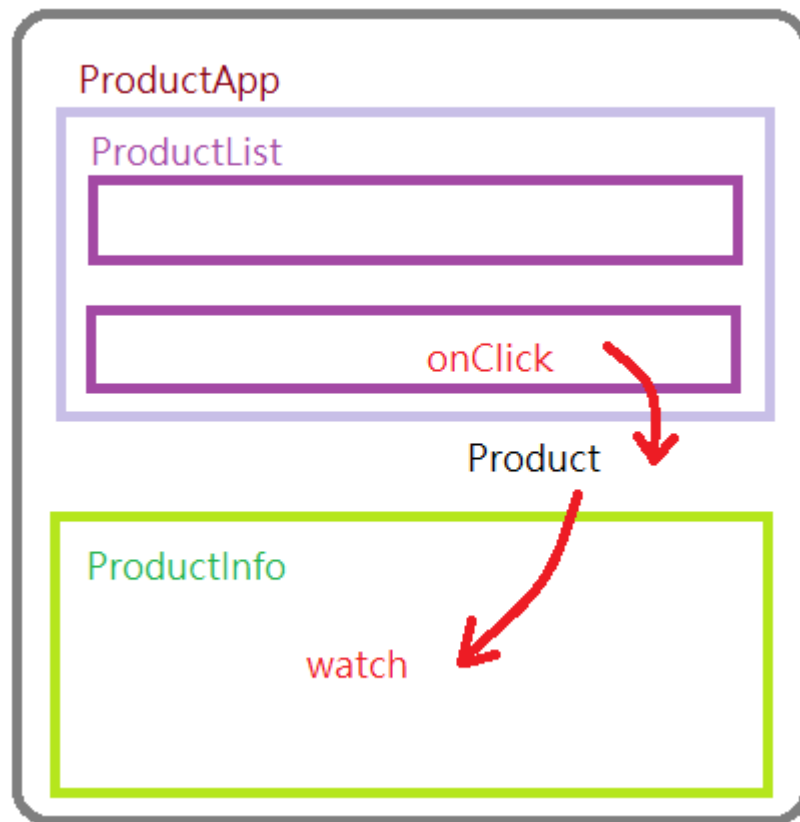
```
{  
  ...  
  watch: {  
    item(value) {  
      this.product = value;  
    },  
  },  
}
```

將 `ProductInfo` 註冊到 `ProductApp` component 底下，並使用該 component 標籤

```
<template>  
  ...  
  <product-info v-if="product" :item="product"></product-info>  
  ...  
</template>  
  
<script>  
  ...  
  import ProductInfo from '@components/ProductInfo';  
  
  export default {  
    data() {  
      return {  
        product: undefined,  
      };  
    },  
    components: {  
      ProductList,  
      ProductInfo,  
    },  
  };  
</script>
```

這裡我們使用 `v-if` 來控制當商品尚未被選擇時的情況隱藏商品詳情區塊，並透過 `v-bind` 的方式，讓商品資料動態地傳入 `ProductInfo` component 中

再來，最重要的一步，我們要偵測使用者在 `ProductList` 點擊商品，並將該商品資料傳送給 `ProductInfo`，圖片示意：



1. 在 ProductList 透過 onClick 事件觸發 (\$emit) 外層的 ProductApp 的 product-click 事件
2. 在 ProductList 的 product-click 事件中改變 ProductApp data 值
3. data 改變，導致 ProductInfo 的 props 值改變
4. ProductInfo 的 watch 偵測到 props 值改變，觸發 ProductInfo 的 data 改變
5. ProductInfo 的 data 更新，頁面內容重新渲染

```
<!-- PrdocutList.vue -->
<template>
  <div>
    <div class="product-container" v-for="product in products"
      :key="product.id" @click="onProductClick(product)">
      ...
    </div>
  </div>
</template>
<script>
export default {
  ...
  methods: {
    onProductClick(product) {
      this.$emit('product-click', product);
    },
  },
}
</script>
```

在 ProductList 的 container 加入 @click 事件監聽，並傳入 product 作為參數，再利用 this.\$emit(eventname, argument) 來觸發外部事件

@click 為 v-on:click 的縮寫

this.\$emit(eventname, argument)，eventname 與外部使用 component 時的事件監聽一致，以 **kebab-case** 格式命名，如 my-event

```
<!-- ProductApp -->
<template>
  <div>
    <product-list v-on:product-click="onProductClick"></product-list>
    ...
  </div>
</template>
<script>
export default {
  ...
  methods: {
    onProductClick(product) {
      this.product = product;
    },
  },
}
</script>
```

在 ProductApp 的 product-list element 加入 @product-click 事件監聽並指定其觸發的方法

如此，我們已經透過傳遞資料的方式做到組件間資料溝通的實作

輸入欲購買商品數量

在商品詳情內容中，還需要提供使用者輸入購買數量的表單欄位、計算金額資訊

接著我們實作表單控制，在 ProductInfo.vue 加上表單控制的區塊，並在 data 中加入輸入項對應的屬性

```
<template>
  <div>
    <h2>Product Info</h2>
    <div class="product-container">
      ...
      <div class="purchase-form">
        <label for="qty">Qty: </label>
        <input type="number" id="qty" v-model="quantity">
      </div>
    </div>
  </div>
</template>
<script>
export default {
  ...
  data() {
    return {
      ...
    }
  }
}
```

```

        quantity: 0,
      };
    },
    ...
  }
</script>

<style scoped>
...

.purchase-form {
  display: flex;
  flex-wrap: nowrap;
  width: 100%;
  justify-content: center;
  align-items: baseline;
}

.form-control {
  display: flex;
  justify-content: flex-end;
  width: 50%;
  padding: 10px;
}

.form-control label {
  font-size: 1.2em;
}

.form-control input::-webkit-inner-spin-button{
  -webkit-appearance: none;
}

.form-control input {
  margin: 0 5px;
  border: 1px solid #ddd;
  box-sizing: border-box;
  border-radius: 2px;
  line-height: 20px;
  width: 40px;
  font-size: 1.1em;
  text-align: center;
}
</style>

```

在 input 使用 v-model 來達到資料綁定，當使用者更改購買數量，data.quantity 也會跟著變動

可以在 template 中加入下面內容驗證

```
<span>{{ quantity }}</span>
```

接著實作小計計算，這裡可以用 computed 來快速地達成我們的需求

```

<script>
export default {
  ...
  computed: {
    amount() {
      return this.product.price * this.quantity;
    },
  },
  ...
}
</script>

```

template 可以如下變更，別忘記加入 filter 來顯示幣值符號

```

<template>
  <div>
    <h2>Product Info</h2>
    <div class="product-container">
      ...
      <div class="purchase-form">
        <div class="form-control">
          <label for="qty">Qty: </label>
          <input type="number" id="qty" v-model="quantity">
        </div>
        <div class="purchase-amount">{{ amount | currency }}</div>
      </div>
    </div>
  </div>
</template>

<style scoped>
  ...

  .purchase-amount {
    display: flex;
    justify-content: flex-start;
    width: 50%;
  }
</style>

```

利用 **Router** 切換頁面內容

雖然已經完成了商品詳情內容，但電商網頁一般都會以另一個頁面顯示商品內容，現在我們透過 Router 來完成 SPA 頁面切換

在 main.js 加入 router 的註冊

```
import Vue from 'vue';
import router from './router';

import ProductApp from './ProductApp';

...

new Vue({
  router,
  render: h => h(ProductApp),
}).$mount('#app');
```

接著編輯 router.js 來進行 router 對應設定，我們可以看到在 vue-cli 建立專案後，routes 底下已經有兩種定義方式

```
export default new Router({
  mode: 'history',
  base: process.env.BASE_URL,
  routes: [
    {
      path: '/',
      name: 'home',
      component: Home,
    },
    {
      path: '/about',
      name: 'about',
      // route level code-splitting
      // this generates a separate chunk (about.[hash].js) for this route
      // which is lazy-loaded when the route is visited.
      component: () => import(/* webpackChunkName: "about" */ './views/About.vue'),
    },
  ],
});
```

第二種定義方式在註解中已經說明，當造訪頁面時才真正建構並讀取該 component 實體

route 物件需要定義三個屬性

- path：網址路徑
- name：路由名稱
- component：對應的 component 實體
- params：欲傳遞的參數
- props：欲傳遞的 props 內容

現在將 routes 更改為符合 ProductApp 的設定，並預設導向商品列表頁

```
export default new Router({
  mode: 'history',
  base: process.env.BASE_URL,
  routes: [
```

```

    {
      path: '/product',
      name: 'product',
      component: () => import('./components/ProductList.vue'),
    },
    {
      path: '/product/:id',
      name: 'productInfo',
      component: () => import('./components/ProductInfo.vue'),
    },
    { path: '*', redirect: '/product' },
  ],
});

```

商品詳情頁使用 `/product/{param}` 來傳遞商品id，因此後面還必須實作利用商品 id 取得商品的服務

修改 ProductApp component，因為已經改用 Router 方式，將不需要的方法及內容移除，僅留下 template

```

<template>
  <div id="productApp">
    <h1>Product App!</h1>
    <router-view />
  </div>
</template>

```

router-view 即是作為 router 控制的顯示內容，當 router 變更時，router-view 會顯示為 router 對應的 component 內容

接著更改 ProductList 按下商品後的觸發事件，在 onProductClick 方法改為使用 router 導頁

```

<script>
export default {
  ..
  methods: {
    onProductClick(product) {
      this.$router.push({ name: 'productInfo', params: { id: product.id } });
    },
  },
}
</script>

```

`$router.push` 即為導頁的實作方法，在傳入的 options 中放入欲導頁的 router 名稱，以及 queryParams 物件，這裡只需要傳入商品 ID 即可

接著在 ProductInfo 加入回到上一頁的功能以回到商品列表頁

```

<template>
  <div>
    <button class="link" @click="onBackClick">
      <i class="arrow-left"></i> Back
    </button>
    ...
  </div>

```

```

    </div>
</template>

<script>
export default {
  ...
  methods: {
    onBackClick() {
      this.$router.back();
    },
  },
}
</script>

<style scoped>
.link {
  margin: 5px;
  display: inline-block;
  padding: 6px 12px;
  font-size: 14px;
  text-align: center;
  text-decoration: underline;
  border: none;
  font-weight: 400;
  color: #337ab7;
  background-color: #fff;
}

.link:hover {
  cursor: pointer;
}

.arrow-left {
  border: solid;
  border-width: 0 2px 2px 0;
  display: inline-block;
  padding: 3px;
  -webkit-transform: rotate(135deg);
  transform: rotate(135deg);
}

...
</style>

```

利用 `$router.back()` 來達到回到上一頁的目的

儲存後在網址後方加入 `/product` 來導至商品列表頁，並按下商品項目進入商品詳情頁，此時因為無法取得商品內容，商品詳情還無法正常顯示，但我們已經完成了透過 **Router** 來切換頁面的功能

利用 **service** 處理資料服務

前面改用 Router 來完成導頁到商品詳情頁，因此資料需要另外在商品詳情頁取得，我們將取得資料服務的邏輯分離出來，利用 service 來處理資料

在 src 底下新增 services 資料夾，並在資料夾中加入 productService.js 檔案

```
import products from '@assets/data/product';

export default {
  get(id) {
    if (id) {
      return products.find(product => product.id === Number(id));
    }
    return products;
  },
};
```

由於資料來源改由 local 的 json 取得，需要另外再新增一個 local json 存放位置，在 src/assets 底下新增 data 資料夾，並加入 product.json 檔案

```
[
  {
    "id": 1,
    "name": "Car",
    "imgUrl": "https://images.pexels.com/photos/1200458/pexels-photo-1200458.jpeg?auto=compress&cs=tinysrgb&h=350",
    "desc": "A luxury car you won't miss it.",
    "price": 199.99
  },
  {
    "id": 2,
    "name": "Bike",
    "imgUrl": "https://images.pexels.com/photos/1239460/pexels-photo-1239460.jpeg?auto=compress&cs=tinysrgb&h=350",
    "desc": "A durable bike you've never ride.",
    "price": 25.00
  }
]
```

再來修改 ProductInfo.vue，並移除原有透過 props 的相關設置，資料取得需要在 created 處理

```
<script>
import productService from '@services/product-service';

export default {
  name: 'ProductInfo',
  data() {
    return {
      product: {},
      quantity: 0,
    };
  },
  created() {
```

```

    this.product = productService.get(this.$route.params.id);
  },
  computed: {
    amount() {
      return this.product.price * this.quantity;
    },
  },
  methods: {
    onBackClick() {
      this.$router.back();
    },
  },
};
</script>

```

因為已經透過 service 來處理資料，商品列表頁也可以使用 ProductService 來取得商品列表

修改 ProductList.vue

```

<script>
import productService from '@/services/product-service';

export default {
  name: 'ProductList',
  data() {
    return {
      products: [],
    };
  },
  created() {
    this.products = productService.get();
  },
  methods: {
    onProductClick(product) {
      this.$router.push({ name: 'productInfo', params: { id: product.id } });
    },
  },
};
</script>

```

因為加入 Router，我們可以進一步區分 components，將 components 區分為兩種

1. views：透過 router 切換的 components
2. components：多個 components 組成頁面，彼此間透過 props、event 方式傳遞

可以把 views 當成父 component，而一個 view 中由多個 components 組成

以上述概念區分，ProductInfo 及 ProductList 就可以視作 views，因此我們將這兩個檔案移到 views 資料夾底下

到此，我們完成了基本的 Vue 實作，後續會再加入 rxjs 讓資料以 observer 方式進行資料連動

Unit test

實際開發還是要先寫 *test code* 再寫 *production code*

現在來撰寫單元測試，利用 `jest` 來進行

關於 `jest` 的文件參考：<https://vue-test-utils.vuejs.org/api/#mount>

ProductApp.vue

先撰寫 `ProductApp` 的測試，在根目錄下找到 `tests/unit` 資料夾，在裡面新增 `productApp.spec.js` 檔案

測試程式使用 `jest` 的 `shallowMount` 幫助我們建立實體，因為其中包含 `RouterView` component，使用 `stubs` 模擬 `router-view`

```
import { shallowMount } from '@vue/test-utils';

import ProductApp from '@/ProductApp';

describe('ProductApp.vue', () => {
  it('renders a message', () => {
    const wrapper = shallowMount(ProductApp, {
      stubs: ['router-view'],
    });

    expect(wrapper.text()).toMatch('Product App!');
  });
});
```

這裡確認 `ProductApp.vue` 有成功掛載並確認文字內容顯示正確

開啟終端機輸入 `npm run test:unit` 來驗證測試是否通過

```
> vue-tutorial@0.1.0 test:unit C:\Project\vue-tutorial
> vue-cli-service test:unit

PASS tests/unit/productApp.spec.js
  ProductApp.vue
    ✓ should renders a message (18ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.852s, estimated 2s
Ran all test suites.
```

ProductList.vue

接著測試 `ProductList.vue`，在 `./tests/unit` 底下新增 `views` 資料夾，並新增 `ProductList.spec.js` 檔案

分別測試：

- created 後的資料變動

由於在 ProductList 中使用到 productService 模組，可以使用 jest.mock 模擬該模組內容，並且建立模擬商品資料

```
const mockProducts = [
  {
    id: 1,
    name: 'Product I',
    imgUrl: 'https://images.pexels.com/photos/1200458/pexels-photo-1200458.jpeg?auto=compress&cs=tinysrgb&h=350',
    desc: 'Description',
    price: 20.00,
  },
];

jest.mock('@services/product-service', () => ({
  get: () => mockProducts,
}));
```

也使用到 currency filter，因此需要註冊模擬的 currency filter

```
vue.filter('currency', value => value);
```

最後測試內容如下

```
import { shallowMount } from '@vue/test-utils';

import ProductList from '@views/ProductList';
import Vue from 'vue';

const mockProducts = [
  {
    id: 1,
    name: 'Product I',
    imgUrl: 'https://images.pexels.com/photos/1200458/pexels-photo-1200458.jpeg?auto=compress&cs=tinysrgb&h=350',
    desc: 'Description',
    price: 20.00,
  },
];

jest.mock('@services/product-service', () => ({
  get: () => mockProducts,
}));

vue.filter('currency', value => value);

describe('ProductList.vue', () => {
  it('sets the correct data', () => {
    const wrapper = shallowMount(ProductList);
```

```
    expect(wrapper.vm.products).toEqual(mockProducts);
  });
});
```

- methods 內的方法

methods 內的 onProductClick 由於使用到 router，所以需要先模擬 router 模組，又因為 router 模組是在 main.js 時註冊，所以需要使用 localVue 來註冊

```
const localVue = createLocalVue();
localVue.use(VueRouter);
const routes = [
  {
    path: '/product/:id',
    name: 'productInfo',
    component: ProductInfo,
  },
];
const router = new VueRouter({ routes });
```

最後測試 onProductClick 方法是否將路由導至 productInfo，由於商品項目是動態渲染的，需要透過 \$nextTick() 的 callback 才有辦法取得渲染後的商品 element

測試應該專注於使用者與介面的互動，因此使用 element.trigger 方式觸發方法而不是直接使用 wrapper.vm.methodName

```
const wrapper = shallowMount(ProductList, { localVue, router });
wrapper.vm.$nextTick(() => {
  wrapper.find('.product-container').trigger('click');
  expect(wrapper.vm.$route.name).toMatch('productInfo');
});
```

完整測試碼

```
import { shallowMount, createLocalVue } from '@vue/test-utils';
import Vue from 'vue';
import VueRouter from 'vue-router';

import ProductList from '@views/ProductList';
import ProductInfo from '@views/ProductInfo';

const mockProducts = [
  {
    id: 1,
    name: 'Product 1',
    imgUrl: 'https://images.pexels.com/photos/1200458/pexels-photo-1200458.jpeg?auto=compress&cs=tinysrgb&h=350',
    desc: 'Description',
    price: 20.00,
  },
];
```

```

];

jest.mock('@services/product-service', () => ({
  get: () => mockProducts,
}));

vue.filter('currency', value => value);

describe('ProductList.vue', () => {
  it('sets the correct data', () => {
    const wrapper = shallowMount(ProductList);

    expect(wrapper.vm.products).toEqual(mockProducts);
  });

  it('trigger onProductClick method', () => {
    const localVue = createLocalVue();
    localVue.use(VueRouter);
    const routes = [
      {
        path: '/product/:id',
        name: 'productInfo',
        component: ProductInfo,
      },
    ];
    const router = new VueRouter({ routes });

    const wrapper = shallowMount(ProductList, { localVue, router });
    wrapper.vm.$nextTick(() => {
      wrapper.find('.product-container').trigger('click');
      expect(wrapper.vm.$route.name).toMatch('productInfo');
    });
  });
});

```

ProductInfo.vue

測試 ProductList.vue，在 ./tests/unit/views 底下新增 ProductInfo.spec.js 檔案

分別測試：

- created 後的資料更動

使用到 ProductService，引用方法參考 ProductList，因為需要取得的商品資料需要依據路由的 id 判斷，在這裡需要針對路由處理

```

const $route = {
  path: '/product/1',
  params: { id: 1 },
};

```

在建立實體時加入模擬的 \$route

```
const wrapper = shallowMount(ProductInfo, { mocks: { $route } });
```

完整測試碼

```
import { shallowMount } from '@vue/test-utils';
import Vue from 'vue';

import ProductInfo from '@views/ProductInfo';

const $route = {
  path: '/product/1',
  params: { id: 1 },
};

const mockProduct = {
  id: 1,
  name: 'Product I',
  imgUrl: 'https://images.pexels.com/photos/1200458/pexels-photo-1200458.jpeg?auto=compress&cs=tinysrgb&h=350',
  desc: 'Description',
  price: 20.00,
};

jest.mock('@services/product-service', () => ({
  get: () => mockProduct,
}));

Vue.filter('currency', value => value);

describe('ProductInfo.vue', () => {
  it('sets the correct data', () => {
    const wrapper = shallowMount(ProductInfo, { mocks: { $route } });

    expect(wrapper.vm.product).toEqual(mockProduct);
  });
});
```

- computed 內的屬性

```
it('compute correct amount', () => {
  const input = wrapper.find('#qty');
  input.element.value = 2;
  input.trigger('input');

  expect(wrapper.vm.amount).toBeCloseTo(40.00);
});
```

因為 wrapper 在每個測試方法中都是用同樣的方式實體化，可以用 beforeEach 減少在每個方法都要加入實體化的程式碼，完整測試碼：

```

import { shallowMount } from '@vue/test-utils';
import Vue from 'vue';

import ProductInfo from '@views/ProductInfo';

const $route = {
  path: '/product/1',
  params: { id: 1 },
};

const mockProduct = {
  id: 1,
  name: 'Product 1',
  imgUrl: 'https://images.pexels.com/photos/1200458/pexels-photo-1200458.jpeg?auto=compress&cs=tinysrgb&h=350',
  desc: 'Description',
  price: 20.00,
};

jest.mock('@services/product-service', () => ({
  get: () => mockProduct,
}));

Vue.filter('currency', value => value);

describe('ProductInfo.vue', () => {
  let wrapper;

  beforeEach(() => {
    wrapper = shallowMount(ProductInfo, { mocks: { $route } });
  });

  it('sets the correct data', () => {
    expect(wrapper.vm.product).toEqual(mockProduct);
  });

  it('compute correct amount', () => {
    const input = wrapper.find('#qty');
    input.element.value = 2;
    input.trigger('input');

    expect(wrapper.vm.amount).toBeCloseTo(40.00);
  });
});

```

- methods 內的方法

由於無法實際測試回到上一頁的事件，因此僅確認方法是否成功執行

```

it('trigger onBackClick method', () => {
  wrapper.setMethods({ onBackClick: jest.fn() });

  wrapper.find('button').trigger('click');

  expect(wrapper.vm.onBackClick).toHaveBeenCalledTimes(1);
});

```

完整測試碼

```

import { shallowMount } from '@vue/test-utils';
import Vue from 'vue';

import ProductInfo from '@views/ProductInfo';

const mockProduct = {
  id: 1,
  name: 'Product I',
  imgUrl: 'https://images.pexels.com/photos/1200458/pexels-photo-1200458.jpeg?auto=compress&cs=tinysrgb&h=350',
  desc: 'Description',
  price: 20.00,
};

jest.mock('@services/product-service', () => ({
  get: () => mockProduct,
}));

const $route = {
  path: '/product/1',
  params: { id: 1 },
};

vue.filter('currency', value => value);

describe('ProductInfo.vue', () => {
  let wrapper;

  beforeEach(() => {
    wrapper = shallowMount(ProductInfo, { mocks: { $route } });
  });

  it('sets the correct data', () => {
    expect(wrapper.vm.product).toEqual(mockProduct);
  });

  it('compute correct amount', () => {
    const input = wrapper.find('#qty');
    input.element.value = 2;
    input.trigger('input');

    expect(wrapper.vm.amount).toBeCloseTo(40.00);
  });
});

```

```
});

it('trigger onBackClick method', () => {
  wrapper.setMethods({ onBackClick: jest.fn() });

  wrapper.find('button').trigger('click');

  expect(wrapper.vm.onBackClick).toHaveBeenCalledTimes(1);
});
});
```

Currency filter

僅須測試回傳值是否正確

```
import Currency from '@/filters/Currency';

describe('Currency.js', () => {
  it('return correct content', () => {
    const input = 10.05;
    expect(Currency(input)).toMatch('$ 10.05');
  });
});
```

附錄

eslint

下面提供 esling 的設定值，在專案根目錄打開 .eslintrc.js 編輯

```
{
  ...
  rules: {
    'no-console': process.env.NODE_ENV === 'production' ? 'error' : 'off',
    'no-debugger': process.env.NODE_ENV === 'production' ? 'error' : 'off',
    'no-alert': process.env.NODE_ENV === 'production' ? 'error' : 'off',
    'linebreak-style': process.env.NODE_ENV === 'production' ? ['error', 'windows'] :
    'off',
    'space-before-function-paren': process.env.NODE_ENV === 'production' ? ['error',
    'never'] : 'off',
    'import/extensions': process.env.NODE_ENV === 'production' ? ['error', 'never'] :
    'off'
  },
  ...
}
```


另外，透過安裝 VS Code 的套件 ESLint 來幫助我們在進行檔案編輯時提示 code style 錯誤，以及儲存時自動以 eslint 設定值調整 code style

1. 安裝 ESLint 套件，並重新開啟 VS Code
2. 按下 `ctrl` + `,` 來開啟使用者設定
3. 在使用者設定頁的右上角，點擊 ... -> Open settings.json
4. 在 User Settings.json 加入

```
{
  ...
  "eslint.validate": [
    {
      "language": "html",
      "autoFix": true
    },
    {
      "language": "vue",
      "autoFix": true
    },
    {
      "language": "javascript",
      "autoFix": true
    }
  ],
  "eslint.autoFixOnSave": true
}
```

Computed Caching vs Methods

比較下面兩個用法

```
<!-- computed -->
<div>{{ reverseMessage }}</div>

<!-- method -->
<div>{{ reverseMessage() }}</div>
```

前者使用 computed 的屬性，後者則是使用 method 方法

```
/* computed */
new Vue({
  computed: {
    reverseMessage: function() {
      return this.message.split('').reverse().join('')
    }
  }
});
```

```

/* method */
new Vue({
  methods: {
    reverseMessage: function() {
      return this.message.split('').reverse().join('')
    }
  }
})

```

差別在於：使用 `computed`，當相依的資料有變更時就會重新計算，而使用 `method` 則只有在重新渲染才重新計算；也就是說，在相依資料沒有變更的情況下，若將關聯的 `elements` 重新渲染，使用 `method` 會需要再重新計算一次，而 `computed` 則不需要

Copmuted vs Watcher

一般情況下，使用 `computed` 比起 `watcher` 更簡潔，如下：

```

new Vue({
  data: {
    firstName: 'Foo',
    lastName: 'Bar',
    fullName: 'Foo Bar'
  },
  watch: {
    firstName: function (val) {
      this.fullName = val + ' ' + this.lastName
    },
    lastName: function (val) {
      this.fullName = this.firstName + ' ' + val
    }
  },
  computed: {
    fullName: function () {
      return this.firstName + ' ' + this.lastName
    }
  }
});

```

當需要較複雜的計算或非同步的方法時，則可以使用 `watcher` 來完成工作