
前言

JavaScript 框架（framework）在現代 Web 前端開發（frontend development）中扮演重要的角色。在開發 Web 專案時，公司選擇框架的原因有很多，包括最終產品的品質、開發成本、編程標準（coding standard）和開發難易度。因此，學習如何使用 JavaScript 框架（如 Vue）對於任何現代 Web 開發人員（或前端開發人員或全端開發人員）來說都是不可或缺的。

本書適用於希望使用 Vue 程式庫、JavaScript 和 TypeScript 從頭到尾學習和開發 Web 應用程式的程式設計師。它完全聚焦於 Vue 及其生態系統如何幫助你以最直接、最舒適的方式建置規模可擴充（scalable）的互動式 Web 應用程式。在介紹基礎知識的同時，我們還將涵蓋用於狀態管理（state management）、測試、動畫、部署和伺服器端描繪（server-side rendering）的 Vue Router 和 Pinia，確保你可以立即著手發展複雜的 Vue 專案。

若你不熟悉 Vue 或 Virtual DOM（虛擬的文件物件模型）的概念，也沒關係。本書不假設你對 Vue 或任何類似框架有任何背景知識。我將從頭開始介紹並指導你學習 Vue 的所有基礎知識。在第 2 章中，我還會帶你了解 Vue 中的 Virtual DOM 概念和反應性系統（reactivity system），作為本書後續內容的基礎。

本書並不要求你懂 TypeScript，但如果你熟悉 TypeScript 基礎知識，就會有更充分的準備。此外，如果你事先掌握了 HTML、CSS 和 JavaScript 的基礎知識，那麼你也能更好地理解本書的內容。在學習任何 Web（或前端）JavaScript 框架之前，打好這三者的堅實基礎都是極為重要的。

歡迎來到 Vue.js 的世界！

Vue.js 最初在 2014 年釋出，受到了快速的採用，特別是在 2018 年。由於其易用性和靈活性，Vue 在開發人員社群中是廣受歡迎的框架。如果你正在尋找一款出色的工具來建置效能卓越的 Web 應用程式並將其交付給終端使用者，Vue.js 就是最佳解答。

本章重點介紹 Vue.js 的核心概念，並引導你了解 Vue.js 開發環境所需的工具。本章也會探討一些實用的工具，使你的 Vue.js 開發過程更易於管理。本章結束時，你將擁有包含簡單 Vue.js 應用程式的工作環境，可以開始學習 Vue.js 的旅程。

Vue.js 是什麼？

Vue.js 或 Vue 在法語中是 **view**（視圖，或稱「檢視」）的意思；它是一種 JavaScript 引擎，用在前端應用程式中建置漸進式（**progressive**）、可組合（**composable**）和反應式（**reactive**）的使用者介面（*user interfaces*，UI）。



從這裡開始，我們將使用 **Vue** 來表示 **Vue.js**。

Vue 是在 JavaScript 的基礎上編寫的，它提供一種有組織的機制來架構並建置 Web 應用程式。它還充當轉換編譯器（**trans-compiler**，*transpiler*），在部署前的建置過程中，將 Vue 程式碼（作為 **Single File Component**，即「單一檔案元件」，我們會在第 59 頁的「Vue 的單一檔案元件結構」中進一步討論）編譯並轉換成等效的 HTML、CSS 和 JavaScript 程式碼。在獨立模式下（配合一個生成的指令稿檔案），Vue 引擎會在執行時期（**run-time**）進行程式碼轉譯。

Vite.js 作為建置者管理工具

Vite.js（或 Vite）於 2020 年推出，是一款 JavaScript 開發伺服器（development server），它在開發過程中使用原生的 ES module⁴ 匯入（import），而不是像 Webpack、Rollup 那樣將程式碼捆裝為 JavaScript 檔案區塊。



從現在起，我們將使用 Vite 來表示 Vite.js。

這種做法能讓 Vite 在開發過程中以極快的速度執行熱重載（hot reload）⁵，使開發體驗流暢無中斷。它還提供許多立即可用的功能，如支援 TypeScript 和視需要編譯（on-demand compilation），這在開發人員社群中正迅速獲得人氣和採用。

Vue 社群已用 Vite 取代了 Vue CLI 工具⁶（在底層使用 Webpack），使其成為建立和管理 Vue 專案的預設建置者工具（builder tool）。

創建一個新的 Vue 應用程式

使用 Vite 有多種方法可以建立新的 Vue 應用程式專案。最直接的方法是在命令提示列或終端機中使用以下命令語法：

```
npm init vue@latest
```

這個命令會先安裝官方的鷹架工具 `create-vue`，然後會列出配置 Vue 應用程式的一串基本問題。

如圖 1-5 所示，本書中 Vue 應用程式使用的組態包括：

Vue 專案名稱，全部為小寫格式

Vite 使用這個值建立內嵌在當前目錄下的新專案目錄。

TypeScript

建立在 JavaScript 基礎上的具型（typed）程式語言。

4 ES modules 是 ECMAScript modules 的縮寫，自 ES6 發行以來，它已成為處理模組（modules）的熱門標準，最初用於 Node.js，近來也用在瀏覽器中。

5 熱重載可自動將新的程式碼變更套用到執行中的應用程式，而無須重啟應用程式或重新整理頁面。

6 Vue command-line interface。

JSX⁷

在第 2 章中，我們將討論 Vue 如何支援以 JSX 標準編寫程式碼（在 JavaScript 程式碼區塊中直接撰寫 HTML 語法）。

Vue Router

在第 8 章中，我們將使用 Vue Router 在應用程式中實作路由（routing）。

Pinia

在第 9 章中，我們將討論如何使用 Pinia 在整個應用程式中管理和共享資料。

Vitest

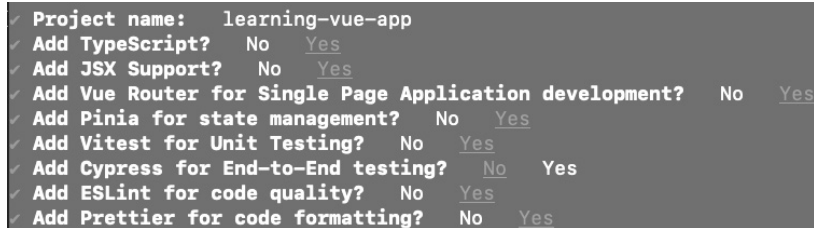
這是所有 Vite 專案的官方單元測試（unit testing）工具，我們將在第 11 章中進一步探討。

ESLint

這個工具會根據一套 ESLint 規則檢查你的程式碼，幫助你遵循編程標準（coding standard），提高程式碼的可讀性，避免隱藏的程式設計錯誤。

Prettier

此工具會自動格式化程式碼的風格，保持程式碼整齊、美觀，並遵循編程標準。



```
✓ Project name:    learning-vue-app
✓ Add TypeScript?  No    Yes
✓ Add JSX Support? No    Yes
✓ Add Vue Router for Single Page Application development? No    Yes
✓ Add Pinia for state management? No    Yes
✓ Add Vitest for Unit Testing? No    Yes
✓ Add Cypress for End-to-End testing? No    Yes
✓ Add ESLint for code quality? No    Yes
✓ Add Prettier for code formatting? No    Yes
```

圖 1-5 新 Vue 應用程式專案的組態

接收所需的組態後，create-vue 會為專案建構相應的鷹架。建立完成後，它將提供一組按順序排列的命令供你執行，以在本地端啟動和執行你的專案（見圖 1-6）。

⁷ JavaScript XML，常用於 React。

Composition API

在上一章中，你學到如何使用傳統的 Options API 來撰寫 Vue 元件。儘管自 Vue 2 以來，Options API 是構成 Vue 元件最常用的 API，但使用 Options API 可能會導致非必要的程式碼複雜性、難以閱讀的大型元件程式碼，並減損它們之間的邏輯可重用性。針對此類用例，本章將介紹另一種構成 Vue 元件的方法，即 Composition API（組合 API）。

在本章中，我們將探索不同的組合掛接器（composition hooks），以便在 Vue 中建立函式型的有狀態元素（functional stateful element）。我們還將學習如何結合 Options API 和 Composition API 來達成更好的反應式控制，並為我們的應用程式編寫自己的可重用（reusable）且可組合（composable）的元素。

使用 Composition API 設定元件

在 Vue 中，使用 Options API 組合元件是一種常見的做法。然而，在許多情況下，我們希望重複使用元件的部分邏輯，而不必擔心像 mixins¹ 中那樣的資料和方法重疊問題，或者希望元件更具可讀性且更有組織。在這種情況下，Composition API 就能派上用場。

在 Vue 3.0 中引進的 Composition API 提供另一種方式，藉助 `setup()` 掛接器（第 66 頁的「setup」）或 `<script setup>` 標記來組合出有狀態的反應式元件。`setup()` 掛接器是元件選項物件（options object）的一部分，會在初始化和建立元件實體之前（在 `beforeCreate()` 掛接器之前）執行一次。

¹ 使用 mixin 時，你就是在撰寫一個新元件的組態。

你只能在此掛接器或等效語法 `<script setup>` 標記中使用 Composition API 函式或可組合掛接器（composables，第 158 頁的「建立可重複使用的可組合掛接器」）。這種結合方式建立了一種有狀態的函式型元件（stateful functional component），為定義元件的反應式狀態和方法、以及初始化其他生命週期掛接器（參閱第 150 頁的「使用生命週期掛接器」）提供絕佳的位置，使程式碼更加直觀易讀。

讓我們從處理元件反應式資料的 `ref()` 和 `reactive()` 函式開始，探索 Composition API 的強大功能。

使用 `ref()` 和 `reactive()` 處理資料

在第 2 章中，我們學習了 Options API 中用於初始化元件資料的 `data()` 函式特性（第 22 頁的「藉由資料特性建立本地狀態」）。從 `data()` 回傳的物件中的所有資料特性都是反應式（reactive）的，這意味著 Vue 引擎會自動觀察宣告的每個資料特性的變化。然而，若有很多資料特性（其中大部分是靜態的）時，這種預設功能可能會為你的元件帶來額外負擔。在這種情況下，Vue 引擎仍會為這些靜態值啟用觀察者（watchers），這是非必要的。為了限制過多資料觀察者的數量，並對要觀察哪些資料特性有更多控制，Vue 在 Composition API 中引進了 `ref()` 和 `reactive()` 函式。

使用 `ref()`

`ref()` 是個函式，它接受單一引數，並回傳以該引數為初始值的反應式物件（reactive object）。我們稱回傳的這個物件為 `ref` 物件：

```
import { ref } from 'vue'

export default {
  setup() {
    const message = ref("Hello World")
    return { message }
  }
}
```

或在 `<script setup>` 中：

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

const message = ref("Hello World")
</script>
```

然後，我們可以在 `script` 區段中透過它單一的 `value` 特性存取回傳物件當前的值。舉例來說，範例 5-1 中的程式碼建立了初始值為 "Hello World" 的反應式物件。

範例 5-1 使用 `ref()` 建立初始值為「Hello World」的反應式訊息

```
import { ref } from 'vue'

const message = ref("Hello World")

console.log(message.value) //Hello World
```



若你搭配使用 `setup()` 掛接器和 Options API，就可以在元件的其他部分存取 `message`，而無須 `.value`，也就是說，只要用 `message` 就夠了。

然而，在 `template` 標記區段，可以不使用 `value` 特性直接獲取其值。例如，範例 5-2 中的程式碼將印出與範例 5-1 相同的 `message`，不過是列印到瀏覽器上。

範例 5-2 在 `template` 區段中存取 `message` 值

```
<template>
  <div>{{ message }}</div>
</template>
<script lang="ts" setup>
import { ref } from 'vue'

const message = ref("Hello World")
</script>
```



`ref()` 函式根據傳入的初始值推斷回傳物件的型別。若要明確定義回傳物件的型別，可以使用 TypeScript 語法 `ref<type>()`，例如 `ref<string>()`。

由於 `ref` 物件是反應式的而且可變（mutable），我們可以為它的 `value` 特性指定新值來更改其值。然後 Vue 引擎就會觸發相關的觀察者並更新元件。

在範例 5-3 中，我們將重新建立 `MyMessageComponent`（來自 Options API 的範例 3-3），它可以接受使用者輸入並改變所顯示的 `message`。

範例 5-3 使用 `ref()` 建立反應式的 `MyMessageComponent`

```
<template>
  <div>
    <h2 class="heading">{{ message }}</h2>
    <input type="text" v-model="message" />
  </div>
</template>
<script lang="ts" setup>
  import { ref } from 'vue'

  const message = ref("Welcome to Vue 3!")
</script>
```

當我們更改輸入欄位的值時，瀏覽器會相應地顯示更新後的 `message` 值，如圖 5-1 所示。

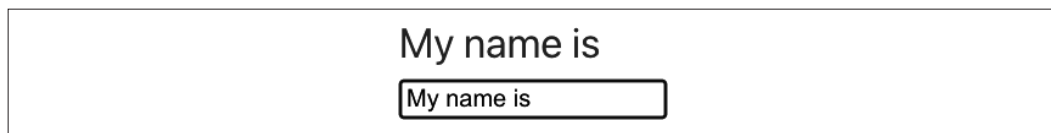


圖 5-1 當我們變更輸入欄位的值時，所顯示的值就會改變

在瀏覽器 Developer Tools 的 Vue 分頁中，我們可以看到 `setup` 區段底下列出 `ref` 物件 `message`，並標有 `Ref`（圖 5-2）。

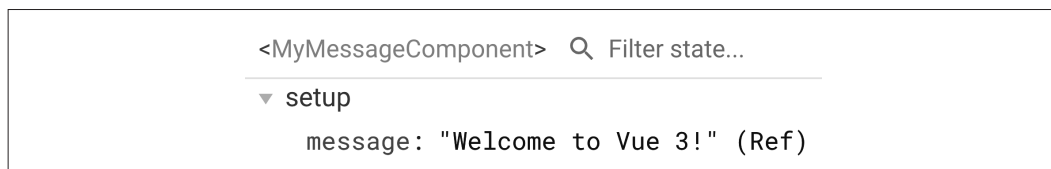


圖 5-2 `ref` 物件 `message` 列在 `setup` 區段底下

如果我們在元件中新增另一個靜態資料 `title`（範例 5-4），Vue 分頁會顯示 `title` 資料特性，但沒有任何標示（圖 5-3）。

範例 5-4 為 `MyMessageComponent` 新增靜態的 `title`

```
<template>
  <div>
    <h1>{{ title }}</h1>
    <h2 class="heading">{{ message }}</h2>
  </div>
</template>
```



```

        <input type="text" v-model="message" />
      </div>
    </template>
    <script lang="ts" setup>
    import { ref } from 'vue'

    const title = "My Message Component"
    const message = ref("Welcome to Vue 3!")
    </script>

```

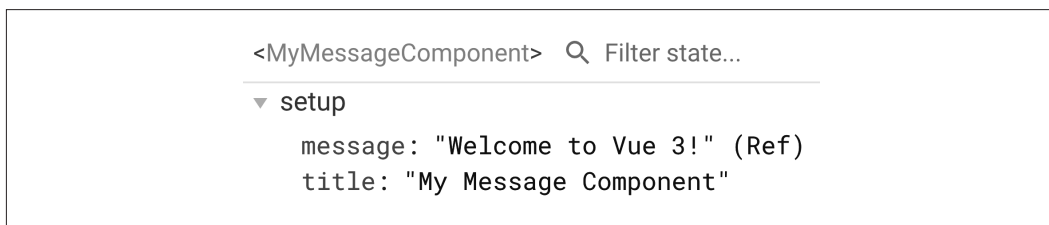


圖 5-3 title 資料特性在列出時沒有任何標示

前面的程式碼（範例 5-4）等同於帶有 `setup()` 掛接器的範例 5-5。

範例 5-5 使用 `setup()` 掛接器建立反應式的 `MyMessageComponent`

```

<template>
  <div>
    <h2 class="heading">{{ message }}</h2>
    <input type="text" v-model="message" />
  </div>
</template>
<script lang="ts">
import { ref } from 'vue'

export default {
  setup() {
    const message = ref("Welcome to Vue 3!")
    return {
      message
    }
  }
}
</script>

```

Vue 中的測試

至此，我們已經學到如何使用不同的 **Vue API** 從零開始開發出完整的 **Vue** 應用程式。我們的應用程式現在已經準備好部署了，但在那之前，我們需要確保我們的應用程式沒有錯誤，可以投入生產。這就是測試（**testing**）的作用。

測試對於任何應用程式的開發都至關重要，因為它有助於在將程式碼釋出到生產環境之前提升對於程式碼的信心和品質。在本章中，我們會學習不同類型的測試以及如何在 **Vue** 應用程式中使用它們。我們還將探索各種工具，如用於單元測試的 **Vitest** 和 **Vue Test Utils**，以及用於端到端（**end-to-end**，**E2E**）測試的 **PlaywrightJS**。

單元測試和 E2E 測試簡介

軟體開發有手動和自動測試兩種實務做法和技巧，以確保應用程式如預期執行。手動測試需要測試人員與軟體進行人工互動，成本可能很高，而自動測試主要是以自動化的方式執行預先定義的測試指令稿（**test script**），其中包含一組測試。自動測試集合可以驗證從簡單到更複雜的應用場景，從單一函式到不同部分的組合。

自動測試比手工測試更可靠、更具規模可擴充性，前提是我們有正確編寫測試，並執行以下測試過程：

單元測試 (Unit testing)

軟體開發中最常見、最低階的測試。我們使用單元測試來驗證執行特定動作的程式碼單元（或程式碼區塊），如函式（**functions**）、掛接器（**hooks**）和模組（**modules**）。我們可以將單元測試與測試驅動開發（**test-driven development**，**TDD**）¹ 結合起來，作為一種標準的開發實務做法。

整合測試 (Integrating testing)

這種類型的測試驗證不同單元程式碼區塊的整合情況。整合測試旨在斷言邏輯函式、元件或模組的流程正確。元件測試（**component testing**）將測試與其內部邏輯整合為單元測試。我們還會模擬（**mock**）大多數上游服務和測試範疇之外的其他函式，以確保測試品質。

端到端 (End-to-end, E2E) 測試

軟體開發中最高階的測試。我們使用 E2E 測試來驗證從客戶端到後端的整個應用程式流程，通常是透過模擬實際的使用者行為。E2E 測試中不會有任何模擬服務或函式，因為我們要測試的是整個應用程式流程。



測試驅動開發（**TDD**）是指首先設計和編寫測試案例（紅色階段），然後修改程式碼以通過測試（綠色階段），最後改善程式碼實作（重構階段）。這有助於在實際開發之前驗證邏輯和設計。

如圖 11-1 所示，這三種測試類型構成了測試金字塔（**pyramid of testing**），其中焦點應該主要放在單元測試上，然後是整合測試，E2E 測試的數量則是最少，因為它主要是為了確保合理性，而且觸發成本可能很高。由於我們建立的應用程式是由任意元件、服務和模組所組成的，因此對每個單獨的函式或功能進行單元測試就足以保證源碼庫（**codebase**）的品質，而且成本和工作量也最少。

作為應用程式測試系統的主要基礎，我們首先使用 **Vitest** 進行單元測試。

¹ 如果你是 TDD 的新手，請從 Saleem Siddiqui 所著的《*Learning Test-Driven Development*》（O'Reilly）開始學習。繁體中文版《Test-Driven Development 學習手冊》由碁峰資訊出版。

使用 Vue Test Utils 測試元件

Vue 引擎使用 Vue 元件的組態來建立和管理瀏覽器 DOM 上的元件實體更新。測試元件意味著我們將測試元件在 DOM 上的描繪結果。我們在 `vite.config.ts` 中將 `test.environment` 設定為 `jsdom`，以模擬瀏覽器環境，這在執行測試的 Node.js 環境中並不存在。我們還使用 `@vue/test-utils` 套件中的 `mount`、`shallowMount` 等方法來幫忙掛載元件，並斷言從虛擬 Vue 節點到 DOM 元素的描繪結果。

我們來看看範例 11-1 中的 `PizzaCard.vue` 元件。

範例 11-1 `PizzaCard` 元件

```
<template>
  <article class="pizza--details-wrapper">
    
    <p>{{ pizza.description }}</p>
    <div class="pizza--inventory">
      <div class="pizza--inventory-stock">Stock: {{ pizza.quantity || 0 }}</div>
      <div class="pizza--inventory-price">$ {{ pizza.price }}</div>
    </div>
  </article>
</template>
<script setup lang="ts">
import type { Pizza } from "@types/Pizza";
import type { PropType } from "vue";

const props = defineProps({
  pizza: {
    type: Object as PropType<Pizza>,
    required: true,
  },
});
</script>
```

我們會建立測試檔案 `tests/PizzaCard.test.ts`，以測試該元件。我們將從 `@vue/test-utils` 匯入 `shallowMount` 方法，以掛載檔案中的元素。`shallowMount` 函式接收兩個主要引數：要掛載的 Vue 元件，以及其中包含掛載元件用的附加資料（如 `prop` 的值、`stubs` 等）的一個物件。下面的程式碼展示了測試檔案看起來的樣子以及 `pizza` `prop` 的初始值：

```
/** tests/PizzaCard.test.ts */
import { shallowMount } from '@vue/test-utils';
import PizzaCard from '@components/PizzaCard.vue';
```

```
describe('PizzaCard', () => {
  it('should render the pizza details', () => {
    const pizza = {
      id: 1,
      title: 'Test Pizza',
      description: 'Test Pizza Description',
      image: 'test-pizza.jpg',
      price: 10,
      quantity: 10,
    };

    const wrapper = shallowMount(PizzaCard, {
      props: {
        pizza,
      },
    });

    expect();
  });
});
```



使用 *shallowMount* vs. 使用 *mount*

shallowMount 方法是包在 *mount* 方法外圍的包裹器 (wrapper)，其 *shallow* 旗標處於啟用狀態。最好使用 *shallowMount* 方法來描繪和測試元件，而無須關心其子元件。若想測試子元件，請使用 *mount* 方法。

shallowMount 方法會回傳一個 Vue 實體 wrapper，其中包含一些輔助方法，讓我們可以模仿 UI 與元件進行的互動。有了這個包裹器實體後，就能編寫我們的斷言。舉例來說，我們可以使用 *find* 方法找到帶有類別 *pizza--details-wrapper* 的 DOM 元素，並斷言其存在：

```
/** tests/PizzaCard.test.ts */
//...

expect(wrapper.find('.pizza--details-wrapper')).toBeTruthy();
```

同樣地，我們可以使用 *text()* 方法斷言 *.pizza--inventory-stock* 和 *.pizza--inventory-price* 元素的文字內容：

```
/** tests/PizzaCard.test.ts */
//...

expect(
  wrapper.find('.pizza--inventory-stock').text()
)
```

我們已經探討了使用 Vitest 和其他工具（如用於 Vue 限定測試的 Vue Test Utils 和程式碼涵蓋率的 Istanbul）進行的單元測試。我們將進入下一個測試層級，學習如何使用 PlaywrightJS 為應用程式編寫 E2E 測試。

使用 PlaywrightJS 進行端到端測試

PlaywrightJS (<https://oreil.ly/sIUKp>)，或稱為 Playwright，是快速、可靠的跨瀏覽器端到端測試框架（end-to-end testing framework）。除 JavaScript 外，它還支援 Python、Java 和 C# 等程式語言。它也支援 WebKit、Firefox 和 Chromium 等多種瀏覽器描繪引擎（rendering engines），使我們能在跨瀏覽器環境中對同一源碼庫執行測試。

要開始使用 Playwright，請執行以下命令：

```
yarn create Playwright
```

Yarn 將執行 Playwright 的建立指令稿，並有提示詢問測試位置（e2e）、是否要安裝 GitHub Actions 作為 CI/CD 的管線工具（pipeline tool），以及是否要安裝 Playwright 瀏覽器。圖 11-18 顯示了在應用程式中初始化 Playwright 的組態範例。

```
#####
#####] 427/427Getting started with writing end-to-end tests with Playwright:
Initializing project in '.'
✓ Where to put your end-to-end tests? · e2e
✓ Add a GitHub Actions workflow? (y/N) · false
✓ Install Playwright browsers (can be done manually via 'yarn playwright install')? (Y/n) · true
Installing Playwright Test (yarn add --dev @playwright/test)...
yarn add v1.22.19
```

圖 11-18 透過提示初始化 Playwright

初始化過程結束後，我們會在專案根目錄下看到新的 e2e 資料夾，其中包含一個 example.spec.ts 檔案。此外，Playwright 還會為我們的專案生成組態檔案 playwright.config.ts，用相關套件修改 package.json，並產生另一個 test-examples 資料夾，其中含有用到 Playwright 的 todo（待辦事項）元件測試範例。

現在就可以在 package.json 中加入新的指令稿命令，使用 Playwright 執行我們的 E2E 測試：

```
"scripts": {
  //...
  "test:e2e": "npx playwright test"
}
```

同樣地，我們可以新增以下命令，為我們的測試產生涵蓋率報告：

```
"scripts": {  
  //...  
  "test:e2e-report": "npx playwright show-report"  
}
```

預設情況下，Playwright 自帶 HTML 涵蓋率報告產生器（coverage reporter），測試執行期間若有任何測試失敗，該報告產生器就會執行。我們可以嘗試使用這些命令執行測試，並檢視通過的範例測試。

檢視 `playwright.config.ts`，看看它包含了什麼：

```
import { defineConfig, devices } from '@playwright/test';  
  
/** playwright.config.ts */  
export default defineConfig({  
  testDir: './e2e',  
  fullyParallel: true,  
  forbidOnly: !!process.env.CI,  
  retries: process.env.CI ? 2 : 0,  
  workers: process.env.CI ? 1 : undefined,  
  reporter: 'html',  
  use: {  
    trace: 'on-first-retry',  
  },  
  projects: [  
    {  
      name: 'chromium',  
      use: { ...devices['Desktop Chrome'] },  
    },  
    {  
      name: 'webkit',  
      use: { ...devices['Desktop Safari'] },  
    },  
  ],  
});
```

組態檔案會匯出由 `defineConfig()` 方法根據一套組態選項所建立的實體，組態選項中帶有下列的主要特性：

testDir

儲存測試的目錄。我們通常在初始化過程中定義它（在我們的例子中為 `e2e`）。

projects

用於執行測試的瀏覽器專案（browser projects）清單。我們可以從相同的 `@playwright/test` 套件匯入 `devices`，並選擇相關的設定來定義供 Playwright 使用的瀏覽器組態，舉例來說，`devices[Desktop Chrome]` 就用於 Chromium 瀏覽器。

worker

執行測試的平行工作者（parallel workers）數量。當我們有許多測試，需要平行執行以加快測試程序時，這個功能會很有幫助。

use

測試執行器的組態物件，包括選擇性的 `baseURL` 作為基礎 URL，以及重試時為失敗的測試啟用追蹤記錄（trace recording）。

其他特性可視需要自訂 Playwright 測試執行器。請參閱 Playwright 說明文件（<https://oreil.ly/nXapE>）中完整的組態選項清單。

我們將保持檔案原樣，並為應用程式編寫我們的第一個 E2E 測試。我們前往 `vite.config.ts`，確保本地伺服器組態如下：

```
//...
export default defineConfig({
  //...
  server: {
    port: 3000
  }
})
```

藉由將通訊埠設定為 3000，我們可以確保本地 URL 始終都會是 `http://localhost:3000`。接下來，我們將在 `e2e` 資料夾中建立新的 E2E 測試檔案，檔名為 `PizzasView.spec.ts`，專門用於測試「/pizzas」頁面。「/pizzas」頁面使用 `PizzasView` 視圖元件顯示披薩清單，其樣板如下：

```
<template>
  <div class="pizzas-view--container">
    <h1>Pizzas</h1>
    <input v-model="search" placeholder="Search for a pizza" />
    <ul>
      <li v-for="pizza in searchResults" :key="pizza.id">
        <PizzaCard :pizza="pizza" />
      </li>
    </ul>
  </div>
```