

國立臺東大學

探訪ASP.NET程式設計

講師：劉韋成 工程師

環境準備

工具	用途	檢查方式
Edge 或 Chrome	執行網頁。F12 開啟開發者工具	—
VS Code 或 Visual Studio	寫程式	code --version
.NET SDK 10	建置和執行 ASP.NET Core	dotnet --version
Node.js 22.12 以上	Vue 專案和前端的測試 (Lab 4、5、6)	node --version
Git	版本控制	git --version
GitHub 帳號	放程式、部署網頁	github.com
Claude Code	Skills 和規格 (Lab 7、8)	claude --version

教材資料夾

labs/	
01-todo-vanilla/	Lab 1 純前端 Todo
02-git-guide/	Lab 2 Git 互動教材
03-todo-mvc/	Lab 3 MVC
04-todo-webapi-vue/	Lab 4 Web API + Vue
05-tests/	Lab 5 測試
06-config/	Lab 6 設定和密碼的管理
07-claude-skills/	Lab 7 Skills
08-spec-first/	Lab 8 先寫規格

每個 Lab 的資料夾裡都是**完成版**。

跟不上的時候，直接執行完成版，再回頭看程式。

同一個 Todo，做三次

Lab 1 純前端



資料存在**瀏覽器**
(localStorage)

```
# 不必啟動  
# 在 index.html 上點兩下  
labs/01-todo-vanilla/index.html
```

Lab 3 MVC



資料存在**伺服器的資料庫**
畫面在**伺服器**組合

```
cd labs/03-todo-mvc/TodoMvc  
dotnet run  
# 開啟 localhost:5058
```

Lab 4 Web API + Vue



資料存在**伺服器的資料庫**
畫面在**瀏覽器**組合

```
# 終端機 1 : 後端  
cd labs/04-todo-webapi-vue/TodoApi  
dotnet run  
# 終端機 2 : 前端 → localhost:5173  
cd labs/04-todo-webapi-vue/todo-web  
npm install  
npm run dev
```

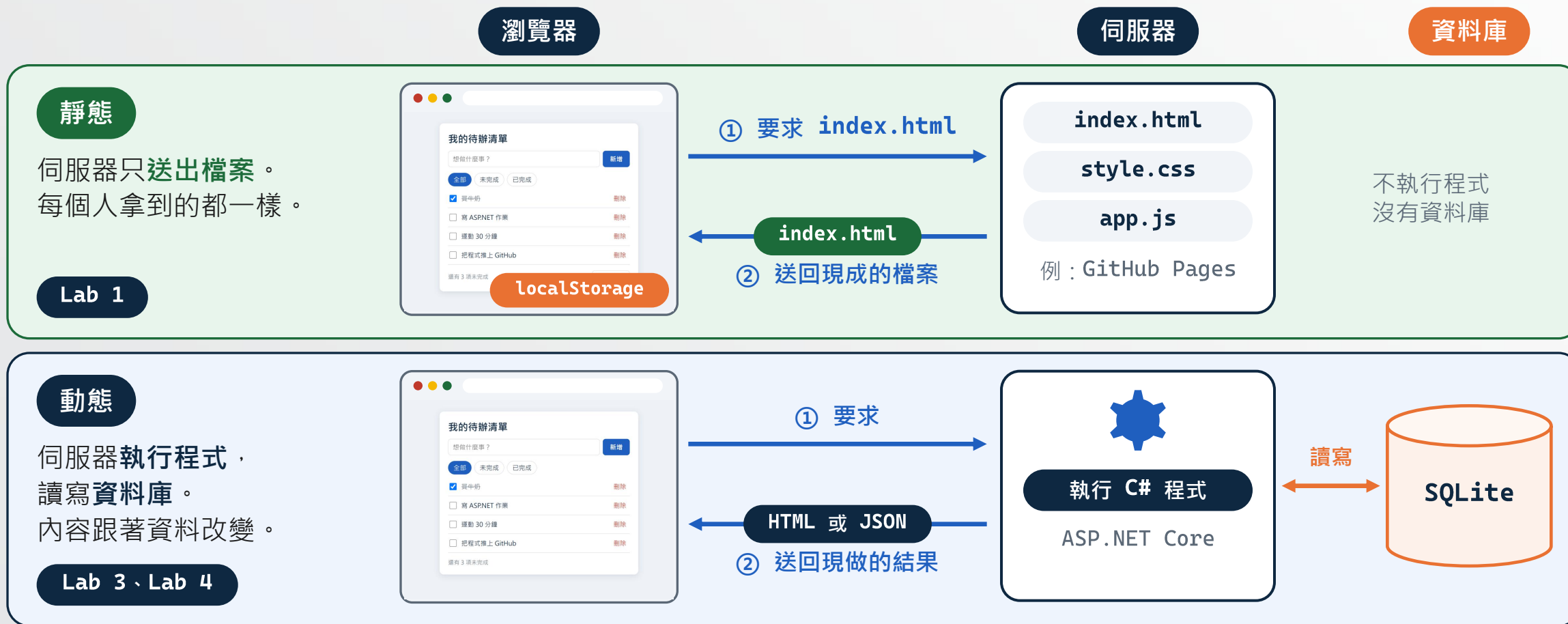
Lab 2 : 把 Lab 1 放上 GitHub。 **Lab 5** : 測試。 **Lab 6** : 設定和密碼。 **Lab 7** : Skills。 **Lab 8** : 規格。

一個網站是怎麼運作的？



- **前端**：在你的瀏覽器裡執行。負責畫面和操作。
- **後端**：在伺服器上執行。負責規則和資料。
- 要求 (**Request**) 和回應 (**Response**) 用 **HTTP** 傳送。
- **Lab 1** 只有前端。 **Lab 3**、**Lab 4** 三個部分都有。

靜態和動態：伺服器做了什麼？



能用靜態完成的事，就用靜態：最簡單。要保存和共用資料，就需要動態。

動態的兩種做法：畫面在哪裡組合？

瀏覽器

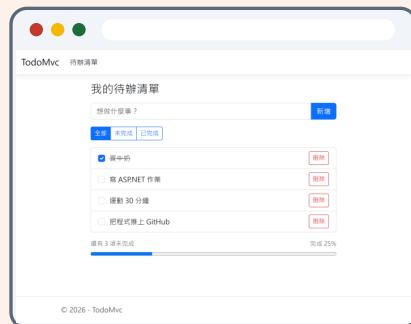
伺服器

資料庫

傳統 server-side render

伺服器組合 HTML。
每次操作都換整頁。

Lab 3 MVC



① 要求頁面

② 完整的 HTML

<html>...</html>

Controller

Model

View (Razor)

★ 畫面在這裡組合

SQLite

前後端分離

伺服器只給資料。
瀏覽器組合畫面，
只換改變的部分。

Lab 4 Web API + Vue



① 下載前端程式 (一次)

② 要資料

③ 只有資料

{ JSON }

前端的檔案

Vue 程式 (靜態)

Web API

只回傳資料，
不組合畫面

SQLite

三種做法，三個 Todo App



LAB 1

純前端 Todo App

HTML · CSS · JavaScript · localStorage



Lab 1: 動手做

LAB 1

- 1 開啟 `labs/01-todo-vanilla`，在 `index.html` 上點兩下
- 2 新增、勾選、篩選、刪除，各試一次
- 3 按 **F5** 重新整理：資料還在嗎？
- 4 按 **F12** > Application > Local Storage，看存了什麼
- 5 把 `style.css` 的 `--blue` 換成你喜歡的顏色

挑戰

讓新增的項目名稱可以編輯修改。

我的待辦清單

想做什麼事？

新增

全部

未完成

已完成

☒ 買牛奶

刪除

☐ 寫 ASP.NET 作業

刪除

☐ 運動 30 分鐘

刪除

☐ 把程式推上 GitHub

刪除

還有 3 項未完成

清除已完成

LAB 2

Git 與 GitHub

commit · branch · push · pull · GitHub Pages · worktree



為什麼需要 Git?

沒有 Git 的時候

報告.docx
報告_修改.docx
報告_最終版.docx
報告_最終版_v2.docx
報告_真的最終版.docx

哪一個才是最新的？
上星期改了什麼？

Git：版本控制的工具

GitHub：放 Git 倉庫的網站

1 存檔點

改壞了，可以回到之前任何一個存檔點。

2 分支

開一個分支試新功能。成功就合併，失敗就丟掉。

3 備份與合作

存檔上傳到 **GitHub**。電腦壞了也不怕，還能多人一起寫。

Git 的四個地方



每天的流程：

git pull → 寫程式 → **git add** → **git commit** → **git push**

git commit: 存檔

DEMO : labs/02-git-guide

```
# 讓 Git 開始管理這個資料夾  
# ( 只做一次 )  
git init
```

```
# 看目前的狀態  
git status
```

```
# 1. 放進暫存區  
git add .
```

```
# 2. 存檔，並寫下說明  
git commit -m "新增標題"
```

```
# 看存檔紀錄  
git log --oneline
```



- **commit** = 一個存檔點
 - 記錄誰、什麼時候、改了什麼
- 存檔分兩步：**add** → **commit**
- 說明要寫清楚「改了什麼」

git branch: 分支

DEMO : labs/02-git-guide

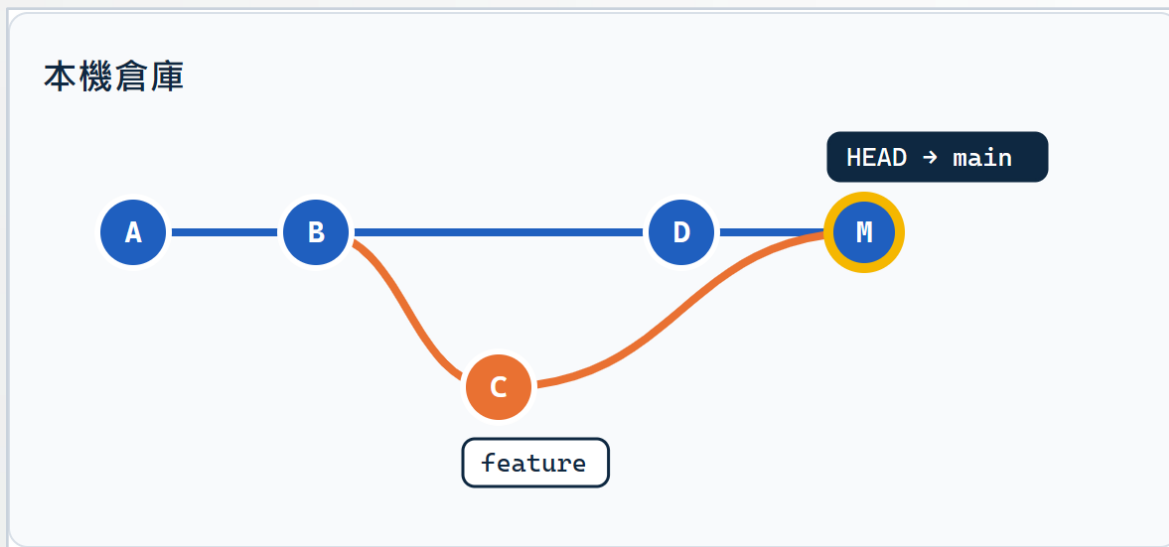
```
# 建立分支，並切換過去
git switch -c feature

# 在 feature 上存檔
# (不影響 main)
git commit -m "新增篩選功能"

# 回到 main
git switch main

# 把 feature 合併進來
git merge feature

# 合併完，刪掉分支
git branch -d feature
```



- 分支 = 貼在 **commit** 上的標籤
- **HEAD** = 你現在的位置
- 新功能在分支上做
 - 做好了才合併回 **main**

git push / git pull: 和 GitHub 同步

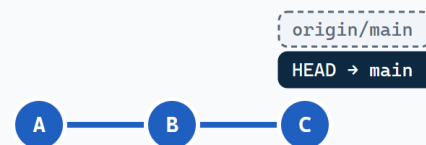
```
# 第一次：設定遠端的網址，然後上傳
git remote add origin https://github.com/{account}/{repo name}.git
git push -u origin main

# 之後：存檔完就上傳
git push

# 開始工作前：先下載別人的新存檔
git pull

# 把 GitHub 上的專案複製到電腦
git clone https://github.com/{account}/{repo name}.git
```

你的電腦 (本機)



GitHub (遠端 origin)



- **push** : 本機 → GitHub
- **pull** : GitHub → 本機
- **origin** : 遠端倉庫的名字
- 先 **pull** 再開始工作
 - 可以減少衝突

GitHub Pages: 把 Lab 1 放上網路

DEMO：現場操作

- 1 在 GitHub 建立 repository (選 **Public**)
- 2 **git push** : 把 Lab 1 推上 **gh-pages** 分支
- 3 GitHub 看到 **gh-pages** , 自動開啟 Pages
- 4 等 1 分鐘 , 開啟網址 : **任何人都看得到**

你的網址

倉庫 `https://github.com/{account}/{repo name}`
網站 `https://{account}.github.io/{repo name}/`

GitHub Pages 只能放**靜態網頁** (HTML 、 CSS 、 JS) 。
它不能執行 C# , 也沒有資料庫 。



- 不用自己架伺服器
- 之後每次 **push** , 網站**自動更新**
 - 改一行 → commit → push → 重新整理

git worktree: 一個倉庫，多個資料夾

```
# 多開一個資料夾，裡面是 main 分支  
git worktree add ../todo-hotfix main
```

```
# 到新資料夾修 bug，然後存檔  
cd ../todo-hotfix  
git commit -am "修正 bug"
```

```
# 列出全部的 worktree  
git worktree list
```

```
# 修完，移除多開的資料夾  
cd ../todo  
git worktree remove ../todo-hotfix
```

📁 todo/

分支：feature

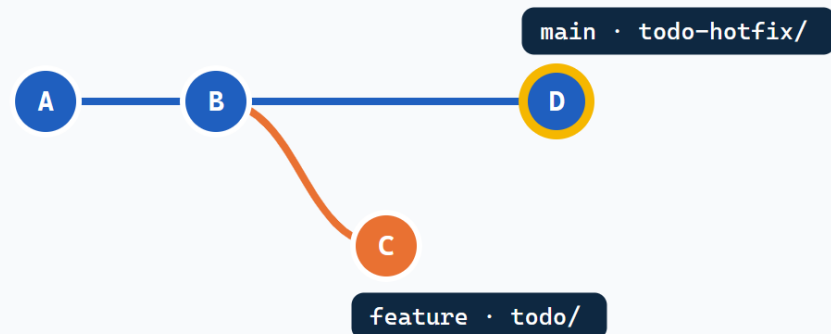
新功能寫到一半，還不能 commit

📁 todo-hotfix/

分支：main

bug 修好了，已經 commit

同一個倉庫



- 新功能寫到一半，突然要修 bug
- 兩個資料夾共用同一個倉庫
- 不必 stash，不必重新 clone

Lab 2: 放上 GitHub Pages

LAB 2

- 1 在 GitHub 建立新的 repository，名稱 **{repo name}** (選 **Public**，其他選項都不要勾)
- 2 在 **labs/01-todo-vanilla** 資料夾開啟終端機，執行下面的指令

```
git init -b gh-pages
git add .
git commit -m "第一次存檔"
git remote add origin https://github.com/{account}/{repo name}.git
git push -u origin gh-pages
```

- 3 GitHub 看到 **gh-pages** 分支，會**自動開啟 Pages**。不必到 **Settings** 設定
- 4 等 1 分鐘，開啟 **<https://{account}.github.io/{repo name}/>**

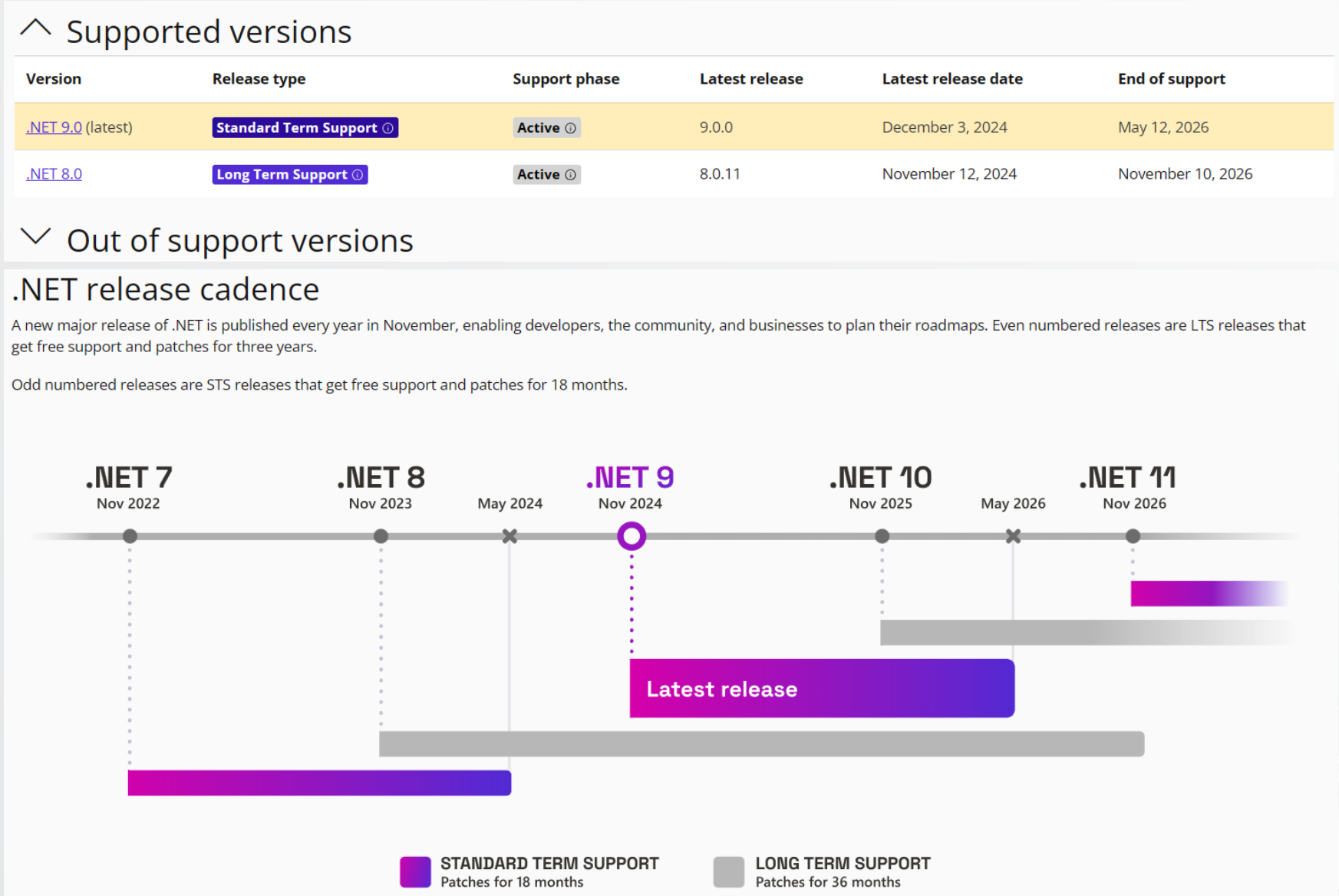
挑戰

改一行程式 → **git add** → **git commit** → **git push**。重新整理網頁，看有沒有更新。

.NET 基本介紹

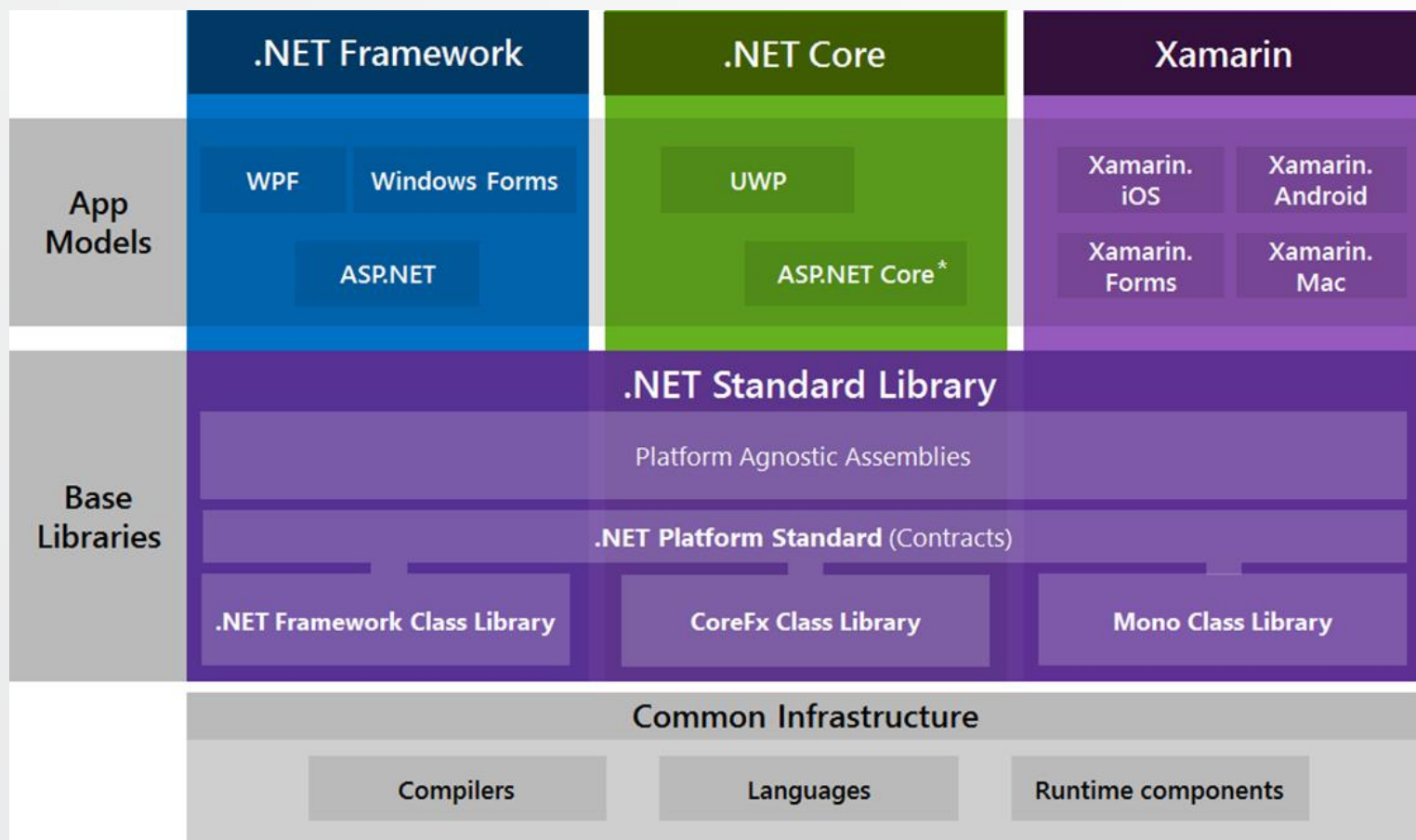


目前的.NET Core 版本



Microsoft 支援的 .NET 實作有四種

- .NET 5.0 與更新版本
- .NET Framework
- Mono
- UWP



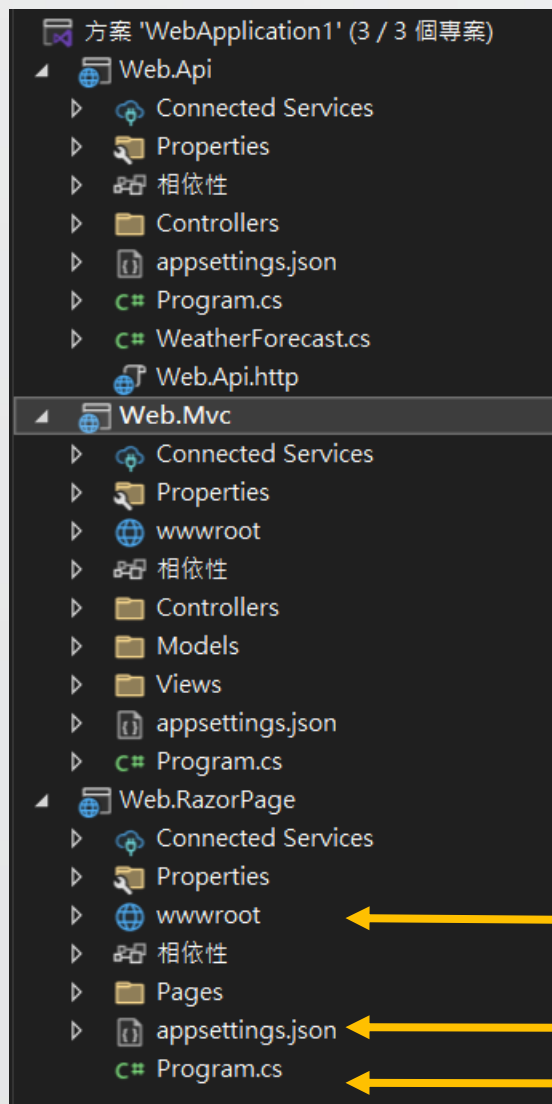
註：.NET 5+也支援WPF、Win Form

安裝 .NET Core SDK & Runtime

- 建置應用程式：.NET Core SDK
- 執行應用程式：.NET Core Runtime
- <https://dotnet.microsoft.com/zh-tw/download/dotnet>

- | | |
|--------------------------|------------------------------|
| – dotnet help | 顯示執行與SDK 命令說明 |
| – dotnet --version | 顯示目前正在使用的SDK 版本 |
| – dotnet --list-runtimes | 顯示已安裝的 .NET Core Runtimes 版本 |
| – dotnet --list-sdks | 顯示已安裝的.NET Core SDK 版本 命令參考 |
| – dotnet --info | 顯示完整的.NET Core 版本資訊 |

認識 ASP .NET Core 網頁開發範本



Web API 開發 HTTP 服務。

Razor Pages 提供了一個清晰分隔考量點的環境，用於開發頁面導向的 Web 應用程式。

MVC 使用 **Model-View-Controller** 設計模式來開發 Web 應用程式。

Blazor 透過可重複使用的 UI 元件進行開發。

網站根目錄

設定檔

程式啟動點

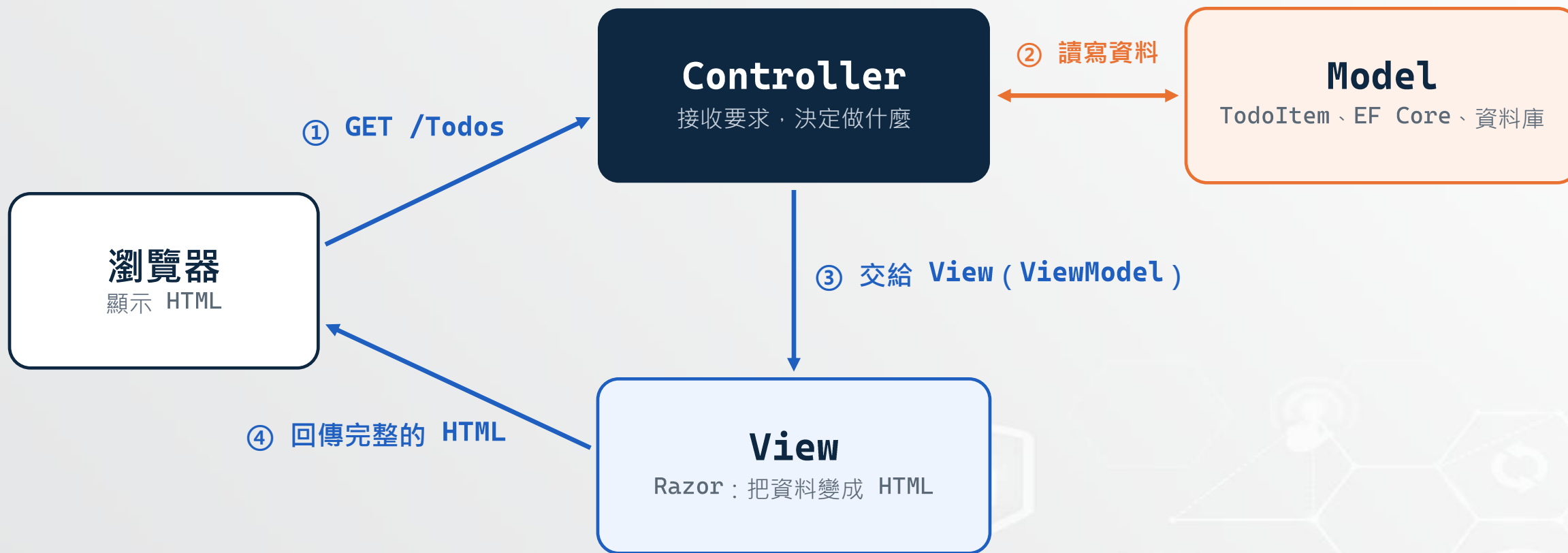
LAB 3

MVC Todo App

ASP.NET Core MVC · Razor · EF Core · SQLite



MVC: 三個角色



View 用 **Razor** 把資料變成 HTML。畫面在**伺服器**組合，瀏覽器只負責顯示。

建立 MVC 專案

```
dotnet new mvc -n TodoMvc
cd TodoMvc
dotnet add package Microsoft.EntityFrameworkCore.Sqlite
dotnet run
```

資料的部分 (Lab 4 也會用到)

- Models/ToDoItem.cs
- Data/ToDoDbContext.cs
- Program.cs 的 AddDbContext
- appsettings.json 的連線字串

畫面的部分 (MVC 的做法)

- Controllers/TodosController.cs
- Views/Todos/Index.cshtml
- Models/ToDoListViewModel.cs
- Models/CreateToDoInput.cs

Program.cs : 預設首頁改成 Todos

```
app.MapControllerRoute(name: "default", pattern: "{controller=Todos}/{action=Index}/{id?}");
```

EF Core: 程式和資料庫之間的翻譯



	我們寫的 C#	EF Core 產生的 SQL	SQLite 的資料表 Todos
新增	<code>db.Todos.Add(todo)</code>	<code>INSERT INTO Todos ...</code>	多一列
讀取	<code>db.Todos.ToList()</code>	<code>SELECT ... FROM Todos</code>	送回全部的列
刪除	<code>db.Todos.Remove(todo)</code>	<code>DELETE FROM Todos ...</code>	少一列

我們只寫 **C#**，不必自己寫 SQL。

EF Core 把 **C#** 翻譯成 SQL，再把結果變回 **C#** 的物件。

Model 與 DbContext

Models/ToDoItem.cs

```
public class ToDoItem
{
    public int Id { get; set; }
    public string Title { get; set; } = "";
    public bool IsDone { get; set; }
}
```

Data/ToDoDbContext.cs

```
public class ToDoDbContext : DbContext
{
    public ToDoDbContext(
        DbContextOptions<ToDoDbContext> options)
        : base(options) { }

    public DbSet<ToDoItem> Todos => Set<ToDoItem>();
}
```

EF Core 自動對應成資料表 **Todos** :

Id	Title	IsDone
1	買牛奶	1
2	寫作業	0

- 類別 ↔ 資料表
- 屬性 ↔ 欄位
- 一個物件 ↔ 一行資料
- **DbContext** : 程式和資料庫之間的窗口
- 不用自己寫 SQL

把 DbContext 交給 DI

Program.cs

```
// 註冊 DbContext : 連線字串放在 appsettings.json
builder.Services.AddDbContext<TodoDbContext>(options =>
    options.UseSqlite(
        builder.Configuration
            .GetConnectionString("TodoDb")));

var app = builder.Build();

// 第一次執行時，建立資料庫檔案和資料表
using (var scope = app.Services.CreateScope())
{
    var db = scope.ServiceProvider
        .GetRequiredService<TodoDbContext>();
    db.Database.EnsureCreated();
}
```

appsettings.json

```
"ConnectionStrings": {
  "TodoDb": "Data Source=todo.db"
}
```

- **DI** : 註冊一次，到處都能用
 - **Controller** 的建構子寫上它就拿得到
- **組態** : 連線字串放設定檔
 - 不寫死在程式裡
- **EnsureCreated**
 - 沒有資料庫就自動建立

EnsureCreated 適合練習。

正式的專案改用 **Migrations** : 資料表可以一版一版更新。

Controller: 準備資料, 交給 View

Controllers/TodosController.cs (節錄)

```
public class TodosController : Controller
{
    // GET /Todos?filter=active : 顯示清單
    [HttpGet]
    public async Task<IActionResult> Index(string filter = "all")
    {
        return View(await BuildViewModel(filter));
    }

    // POST /Todos/Create : 新增一筆
    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Create(CreateTodoInput newTodo)
    {
        if (!ModelState.IsValid)
        {
            return View("Index", await BuildViewModel("all"));
        }

        _db.Todos.Add(new TodoItem { Title = newTodo.Title!.Trim() });
        await _db.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
}
```

← 回傳 View (HTML)

← 防止偽造的表單

← Model Binding

← 驗證沒過：顯示錯誤

← 成功：轉址回清單頁

View: 用 Razor 把資料變成 HTML

Views/Todos/Index.cshtml (節錄)

```
@model TodoListViewModel
```

← 收到的資料型別

```
<form asp-action="Create" method="post">
    <input asp-for="NewTodo.Title" placeholder="想做什麼事?" />
    <button type="submit">新增</button>
    <span asp-validation-for="NewTodo.Title"></span>
</form>
```

← Tag Helper : 產生網址

← 顯示驗證訊息

```
<ul>
    @foreach (var todo in Model.Items)
    {
        <li>
            <form asp-action="Toggle" asp-route-id="@todo.Id" method="post">
                <input type="checkbox" class="js-auto-submit" checked="@todo.IsDone" />
                <span>@todo.Title</span>
            </form>
        </li>
    }
</ul>
```

← 伺服器上的迴圈

```
<span>還有 @Model.Remaining 項未完成</span>
<span>完成 @Model.Percent%</span>
```

← @ : 輸出 C# 的值

表單驗證：伺服器一定要再檢查

Models/CreateTodoInput.cs

```
public class CreateTodoInput
{
    [Required(ErrorMessage = "請輸入待辦事項")]
    [StringLength(100,
        ErrorMessage = "待辦事項最多 100 個字")]
    public string? Title { get; set; }
}
```

- 規則寫在 Model 的屬性上
- Model Binding 時自動檢查
 - 結果放在 ModelState
- 畫面上的限制可以被繞過
 - 伺服器一定要再驗證一次



The screenshot shows the 'TodoMvc' application interface. At the top, it says 'TodoMvc 待辦清單'. Below this is a section titled '我的待辦清單'. There is a text input field with the placeholder '想做什麼事?' and a blue '新增' button. Below the input field, a red error message '請輸入待辦事項' is displayed. Underneath the error message are three buttons: '全部' (selected), '未完成', and '已完成'. Below these buttons is a large empty rectangular box with the text '目前沒有項目' in the center. At the bottom of this section, there is a progress bar showing '還有 0 項未完成' on the left and '完成 0%' on the right. The footer of the application says '© 2026 - TodoMvc'.

沒有輸入就按「新增」：伺服器回傳同一頁，並顯示錯誤訊息。

Lab 3：動手做

LAB 3

- 1 終端機：`cd labs/03-todo-mvc/ToDoMvc`
再執行 `dotnet run`
- 2 新增、勾選、刪除。注意：每次操作，**整頁都重新載入**
- 3 不輸入文字，直接按「新增」：看驗證訊息
- 4 **F12** > Network：伺服器每次都回傳**完整的 HTML**
- 5 按右鍵 > 檢視網頁原始碼
找到 `__RequestVerificationToken`

挑戰

在清單上方顯示「共 N 項」。(要改三個地方：ViewModel、Controller、View)



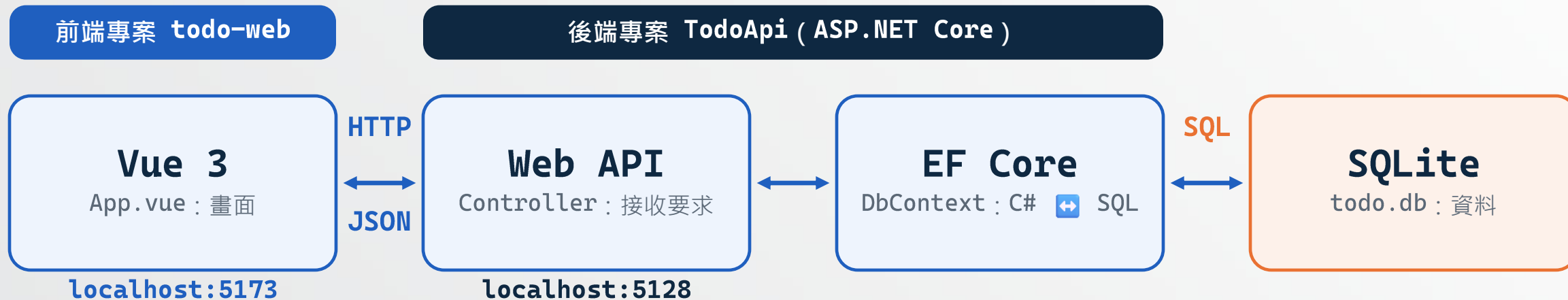
LAB 4

Web API + Vue 3 Todo App

ASP.NET Core Web API · EF Core · SQLite · Vue 3 · Vite



架構：前後端分離



前端和後端的對話

```
GET /api/todos
```

```
200 OK
```

```
[  
  { "id": 1, "title": "買牛奶", "isDone": true },  
  { "id": 2, "title": "寫作業", "isDone": false }  
]
```

- 前端、後端是兩個專案
 - 各自開發、執行、部署
- 後端只回傳資料 (JSON)
 - 畫面由前端的 Vue 組合
- 同一個 API 也能給手機 App 用

兩種做法比一比

	MVC (Lab 3)	Web API + Vue (Lab 4)
專案	一個	兩個：前端專案 + 後端專案
畫面在哪裡組合	伺服器 (Razor)	瀏覽器 (JavaScript)
伺服器回傳	完整的 HTML	JSON 資料
操作之後	整頁重新載入	只更新一部分畫面
要寫的 JavaScript	很少	多
適合	表單為主的系統，例如後台管理	互動多的畫面。API 可給手機 App 共用

沒有哪一個比較好。很多系統兩種一起用。

建立兩個專案：後端和前端

終端機 1：後端 (.NET)

```
dotnet new webapi --use-controllers -n TodoApi
cd TodoApi
dotnet add package Microsoft.EntityFrameworkCore.Sqlite
dotnet run
```

TodoApi/ → http://localhost:5128

Controllers/	接收 HTTP 要求
Models/	資料的樣子
Data/	DbContext
Program.cs	程式的起點
appsettings.json	設定

終端機 2：前端 (Node.js)

```
npm create vue@latest todo-web
cd todo-web
npm install
npm run dev
```

todo-web/ → http://localhost:5173

index.html	網頁的進入點
src/main.js	啟動 Vue
src/App.vue	畫面和行為
src/api.js	呼叫後端
.env	後端的網址

NuGet：.NET 的套件庫。 **npm**：JavaScript 的套件庫。 **SQLite**：資料庫就是一個檔案。
兩個專案**互相不參考**，只用 HTTP 溝通。

REST API：用「動詞 + 網址」操作資料

動作	HTTP 動詞	網址	對應的 SQL	成功時回傳
取得全部	GET	/api/todos	SELECT	200 和清單
新增	POST	/api/todos	INSERT	201 和新資料
修改	PUT	/api/todos/5	UPDATE	204
刪除	DELETE	/api/todos/5	DELETE	204

失敗的時候

400：送來的資料不正確

404：找不到這筆資料

CRUD

Create、Read、Update、Delete

幾乎所有系統都在做這四件事

Controller: 接收要求, 存取資料庫

Controllers/TodosController.cs (節錄)

```
[ApiController]
[Route("api/todos")]
public class TodosController : ControllerBase
{
    private readonly TodoDbContext _db;
    public TodosController(TodoDbContext db) { _db = db; }

    [HttpGet]
    public async Task<List<TodoItem>> GetAll()
    {
        return await _db.Todos.OrderBy(t => t.Id).ToListAsync();
    }

    [HttpPost]
    public async Task<ActionResult<TodoItem>> Create(TodoRequest request)
    {
        var title = TodoTitle.Normalize(request.Title);
        if (title is null) return BadRequest(new { message = TodoTitle.ErrorMessage });
        var todo = new TodoItem { Title = title, IsDone = request.IsDone };
        _db.Todos.Add(todo);
        await _db.SaveChangesAsync();
        return CreatedAtAction(nameof(GetById), new { id = todo.Id }, todo);
    }
}
```

← 網址的開頭

← 建構子注入 (DI)

← GET /api/todos

← SELECT

← POST /api/todos

← INSERT

先測試 API，再寫畫面

TodoApi.http

```
### 新增一筆
POST http://localhost:5128/api/todos
Content-Type: application/json

{
  "title": "買牛奶"
}

### 取得全部
GET http://localhost:5128/api/todos
```

伺服器的回應

```
200 OK
Content-Type: application/json

[
  {
    "id": 1,
    "title": "買牛奶",
    "isDone": false
  }
]
```

- Visual Studio 和 VS Code (REST Client 擴充功能) 可以直接按 **Send Request**
- **GET** 的網址也可以直接用瀏覽器開啟

CORS: 允許前端呼叫後端



TodoApi/Program.cs (節錄)

```
// 前端的網址放在 appsettings.json
var frontendUrl = builder.Configuration["FrontendUrl"]
    ?? throw new InvalidOperationException(
        "appsettings.json 沒有設定 FrontendUrl");

builder.Services.AddCors(options =>
    options.AddDefaultPolicy(policy =>
        policy.WithOrigins(frontendUrl)
            .AllowAnyHeader().AllowAnyMethod()));

var app = builder.Build();
app.UseCors(); // 要放在 UseAuthorization 之前
```

TodoApi/appsettings.json

```
"FrontendUrl": "http://localhost:5173"
```

todo-web/.env

```
VITE_API_URL=http://localhost:5128
```

- 來源 = 協定 + 主機 + port
- port 不同，就是不同的來源
- 兩邊都要知道對方的網址

瀏覽器預設擋下跨來源的讀取。後端用 **CORS** 說明：我允許哪一個前端。

Vue 3：資料變了，畫面自動更新

todo-web/src/App.vue (節錄)

```
<template>
  <form @submit.prevent="addTodo">
    <input v-model="newTitle" type="text">
    <button type="submit">新增</button>
  </form>

  <ul>
    <li v-for="todo in visibleTodos" :key="todo.id">
      <input type="checkbox" v-model="todo.isDone"
        @change="saveTodo(todo)">
      <span>{{ todo.title }}</span>
    </li>
  </ul>

  <span>還有 {{ remaining }} 項未完成</span>
</template>
```

語法	意思
v-model	輸入框和資料保持一樣
v-for	有幾筆資料，就產生幾個元素
@submit	事件發生時，執行方法
{{ }}	把資料顯示出來

App.vue：一個檔案放畫面和行為

Lab 1：自己寫程式 **重畫畫面**

Vue：資料一變，畫面 **自動更新**

Vue 呼叫 API: fetch

todo-web/src/App.vue 的 <script> (節錄)

```
import { TODOS_URL, ensureOk, sendJson } from './api.js';
export default {
  data() {
    return { todos: [], newTitle: '', filter: 'all', error: '' };
  },
  methods: {
    async loadTodos() {
      const response = await fetch(TODOS_URL);
      ensureOk(response);
      this.todos = await response.json();
      this.error = '';
    },
    async addTodo() {
      const response = await sendJson('POST', TODOS_URL, { title: this.newTitle });
      if (response.status === 400) {
        const problem = await response.json();
        this.error = problem.message ?? '資料不正確';
        return;
      }
      ensureOk(response);
      this.newTitle = '';
      await this.loadTodos();
    },
  },
  async mounted() { await this.loadTodos(); },
};
```

← 後端的網址：在 **api.js**

← 畫面用到的資料

← **GET**：讀取清單

← 後端回傳錯誤：丟出錯誤

← 成功：清掉錯誤訊息

← **400**：顯示錯誤訊息

← 改完，重新讀一次

← 一開始先讀資料

Lab 4: 動手做

LAB 4

- 1 後端：`cd labs/04-todo-webapi-vue/ToDoApi`
再執行 `dotnet run`
- 2 前端：`cd labs/04-todo-webapi-vue/todo-web`
再執行 `npm install` 和 `npm run dev`
- 3 瀏覽器開啟 `http://localhost:5173`，新增幾筆資料
- 4 **F12** > Network：要求送到 `localhost:5128`，回應是 JSON
- 5 關掉後端 (`Ctrl+C`)，再 `dotnet run` 一次：資料還在嗎？

挑戰

加上「清除已完成」按鈕。(提示：新增 `DELETE /api/todos/done`，再用 `Vue` 呼叫它)



LAB 5

單元測試與 E2E 測試

Vitest · xUnit · Playwright



為什麼要寫測試？

U 單元測試 (Unit Test)

測一個小功能 (一個函式) 。

不需要網站和資料庫。

很快：幾毫秒。 工具：**Vitest**、**xUnit**

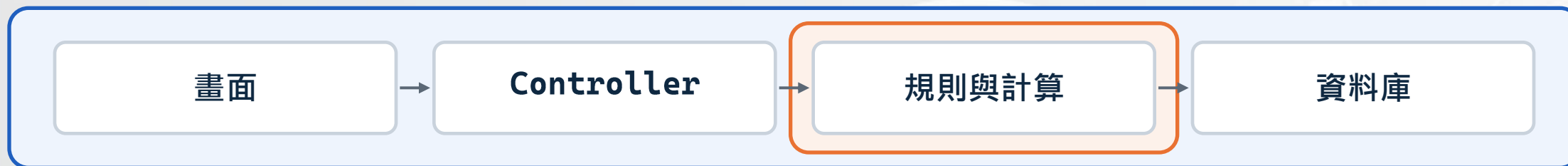
E2E E2E 測試 (End-to-End)

像使用者一樣操作瀏覽器。

一次檢查整條路。

較慢：幾秒。 工具：**Playwright**

E2E 測試：檢查整條路



單元測試：只檢查這一塊

測試 = 把「檢查」寫成程式，讓電腦幫你重複做。

範例：會員註冊表單

會員註冊

身分證字號

A123456789

帳號

amy

註冊

規則 1：身分證字號要正確

1 個英文字 + 9 個數字。

最後一碼是**檢查碼**：用前面 9 碼算出來。

只需要**計算**，不需要資料庫。

規則 2：帳號不可重複

要問**資料庫**：這個帳號有人用了嗎？

只有**後端**能回答。

規則很小，但是**情況很多**。每改一次程式，全部的情況都要再檢查一次。

單元測試：一條規則，很多種情況

輸入	期望的結果	這是什麼情況
A123456789	正確	格式和檢查碼都對
A123456788	錯誤	檢查碼不對（打錯最後一碼）
A12345678	錯誤	少一碼
1234567890	錯誤	第一碼不是英文字
a123456789	正確	小寫也接受：先轉成大寫
（沒有輸入）	錯誤	空白

① 準備 (Arrange)

輸入：A123456788

② 執行 (Act)

呼叫「檢查身分證字號」

③ 驗證 (Assert)

結果應該是「錯誤」

一個測試 = 一種情況。6 個測試，電腦不到 1 秒就檢查完。

用手檢查：每改一次程式，這 6 種情況都要再打一次。

前端的單元測試：規則在瀏覽器裡

前端

瀏覽器

規則用 JavaScript 寫：isValidTaiwanId()

- 使用者一打錯，馬上顯示錯誤
- 不必等伺服器回應
- 測試不需要瀏覽器和伺服器
- 工具：**Vitest** 指令：**npm test**

web-tests/taiwanId.test.js (節錄)

```
import { isValidTaiwanId }
  from '../SignupApp/wwwroot/taiwanId.js'

test('正確的身分證字號', () => {
  expect(isValidTaiwanId('A123456789')).toBe(true)
})

test('檢查碼不對', () => {
  expect(isValidTaiwanId('A123456788')).toBe(false)
})
```

npm test 的結果

```
Test Files  1 passed (1)
Tests       7 passed (7)
Duration    207ms
```

前端的檢查是為了使用者的體驗：馬上知道哪裡打錯。

後端的單元測試：伺服器一定要再檢查

前端的檢查可以被跳過（F12、直接呼叫 API）。所以後端要再檢查一次。

註冊的帳號	期望的結果（已經有 amy）
bob	接受
amy	拒絕：重複
Amy	拒絕：大小寫不同，也算重複
amy	拒絕：先去掉前後的空白

身分證字號：同樣的 6 種情況，後端再測一次。

帳號不可重複：只有後端能檢查。

測試用假的帳號清單，不用真的資料庫。

SignupApp.Tests/Unit/AccountRuleTests.cs (節錄)

```
[Fact]
public void 大小寫不同_也算重複()
{
    // 準備：已經有帳號 amy
    var rule = new AccountRule(["amy"]);

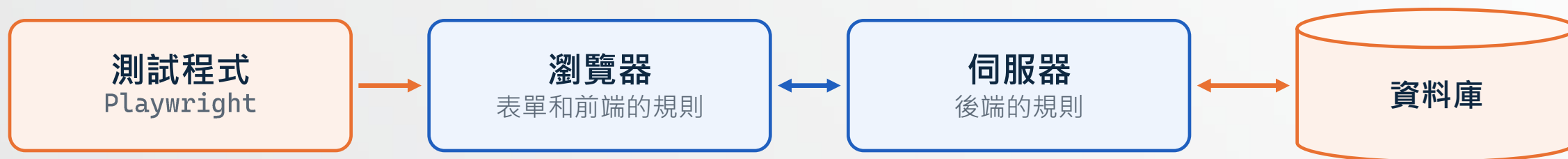
    // 執行：註冊 Amy
    var ok = rule.CanRegister("Amy");

    // 驗證：應該被拒絕
    Assert.False(ok);
}
```

工具：xUnit 指令：dotnet test

後端的檢查是為了保護資料：不正確的資料不可以進資料庫。

E2E 測試：像使用者一樣填表單



1 正確的資料

填好兩個欄位，按註冊。
畫面顯示「**註冊成功**」。

2 身分證字號打錯

欄位旁邊顯示錯誤。
表單**沒有送出**。

3 帳號已經有人用

按註冊之後，畫面顯示
「**這個帳號已經有人使用**」。

E2E 測試檢查：各部分**接起來**之後對不對。
它比較慢（一個要幾秒），所以只測**重要的流程**。
規則的每一種情況，交給單元測試。

三種測試，各檢查什麼？

	前端的單元測試	後端的單元測試	E2E 測試
檢查什麼	瀏覽器裡的規則	伺服器的規則	整個流程
例子	身分證字號	身分證字號、帳號不可重複	填表單，到看到結果
工具	Vitest	xUnit	Playwright
速度	很快（毫秒）	很快（毫秒）	較慢（秒）
數量	多：每種情況一個	多：每種情況一個	少：重要的流程

規則的每一種情況：單元測試。 重要的流程：E2E 測試。

程式由 AI agent 寫。測試讓你知道 agent 有沒有改壞東西。

Lab 5: 動手做

LAB 5

- 1 網站：`cd labs/05-tests/SignupApp`，執行 `dotnet run`
用帳號 `amy` 註冊看看。試完按 `Ctrl+C` 關掉網站
- 2 前端的單元測試：`cd labs/05-tests/web-tests`
執行 `npm install` 和 `npm test`
- 3 後端的單元測試：`cd labs/05-tests/SignupApp.Tests`
執行 `dotnet test --filter Category=Unit`
- 4 E2E 測試：先執行 `$env:E2E_HEADED = "1"`
再執行 `dotnet test --filter Category=E2E`
- 5 改壞規則：把 `taiwanId.js` 最後的 `% 10` 改成 `% 9`
再執行 `npm test`：哪些測試失敗？然後改回來

挑戰

身分證字號的第 2 碼只能是 1 或 2。請 agent 加上這條規則和測試。前端和後端都要改。

會員註冊

身分證字號

A123456789

帳號

amy

這個帳號已經有人使用

註冊

帳號 amy 已經有人使用。

帳號 amy 已經有人使用

LAB 6

參數配置與管理

appsettings · 環境 · User Secrets · 環境變數 · Vite .env



為什麼設定不寫在程式裡？



1 每個環境的值不同

後端的網址、資料庫的位置、寄信的伺服器。
換環境時**只換設定**，程式不用改。

2 密碼不可以進 Git

程式會放上 **GitHub**。
密碼寫在程式裡，等於把鑰匙交給**每個看得到程式的人**。

程式一份，設定多份。密碼和 **API key** 另外放，不進 **Git**。

情境：Todo 網站要上線了

網站需要的設定	例子	種類
網站的名稱	Todo (測試環境)	每個環境不同
前端的網址	localhost:5174	每個環境不同
後端的網址	localhost:5148	每個環境不同
寄信的伺服器 and port	smtp.example.com、587	一般的設定
寄信的密碼		密碼
天氣服務的 API key		密碼

- 同一份程式，在三個環境執行：開發、測試、正式
- 每個環境不同的值，放在哪裡？
- 密碼和 API key，放在哪裡？

開發環境

設定檢視器

前端的設定

來源：.env 檔案。build 的時候就放進 JavaScript。任何人都看得到。不可以放密碼。

名稱	值
MODE	development
VITE_ENV_NAME	開發環境
VITE_APP_TITLE	設定檢視器
VITE_API_URL	http://localhost:5148
WEB_PORT	(讀不到：沒有 VITE_ 開頭)

後端的設定

來源：GET http://localhost:5148/api/settings。密碼只留在後端，前端只拿到遮罩。

後端的環境：Development

名稱	值	來源
Site:Name	Todo (開發環境)	appsettings.Development.json
Site:FrontendUrl	http://localhost:5174	appsettings.json
Mail:Host	localhost	appsettings.Development.json
Mail:Port	2525	appsettings.Development.json
Mail:Password	*****	User Secrets
Weather:BaseUrl	https://api.weather.example	appsettings.json
Weather:ApiKey	*****	User Secrets

工具：設定檢視器 (Lab 6)。它顯示每個設定的值，以及這個值從哪裡來。

三個環境：開發、測試、正式

環境	名稱	誰在用	資料和行為
開發	Development	開發人員，在自己的電腦	假的資料。顯示詳細的錯誤訊息
測試	Staging	測試人員，上線前檢查	假的資料。設定和正式環境接近
正式	Production	真的使用者	真的資料。不顯示錯誤的細節

ConfigApi/Properties/launchSettings.json (節錄)

```
"http": {
  "applicationUrl": "http://localhost:5148",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development" }
},
"staging": {
  "applicationUrl": "http://localhost:5248",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Staging" }
}
```

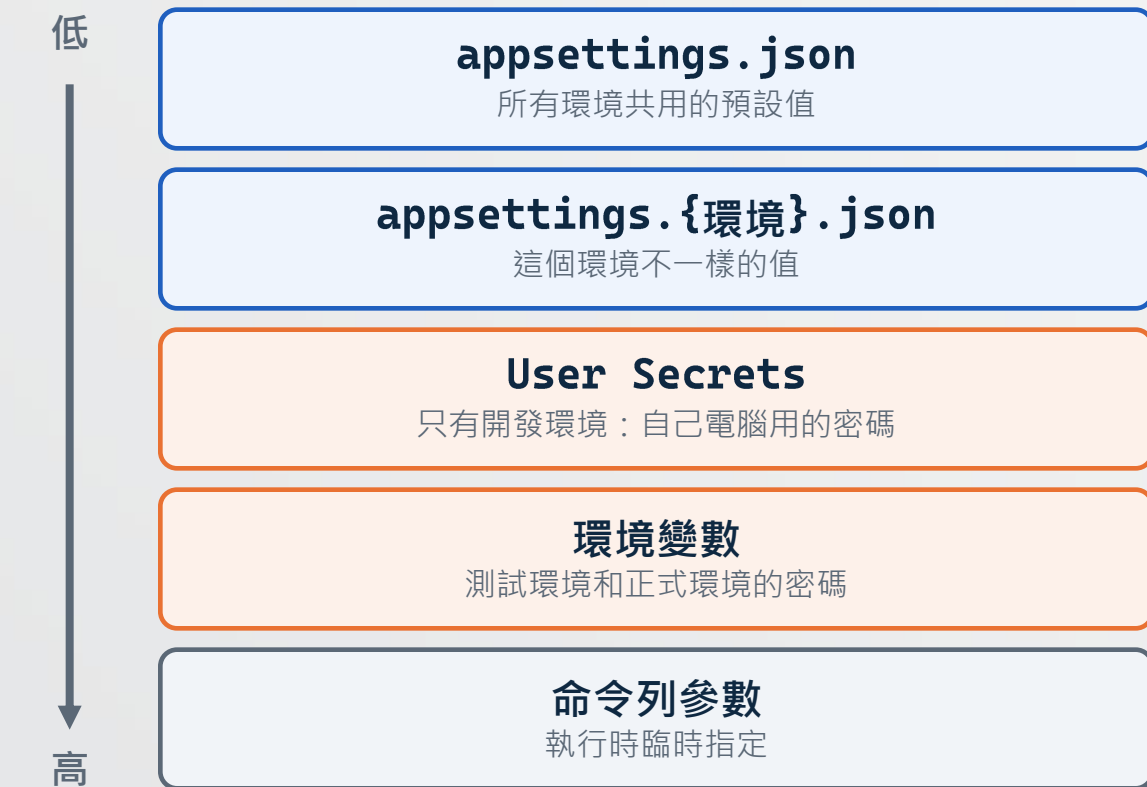
這個 Lab 用另一個 port 模擬測試環境。

終端機

```
# 開發環境 (預設的 profile)
dotnet run
# 測試環境
dotnet run --launch-profile staging
```

- **ASPNETCORE_ENVIRONMENT** 決定環境
- 沒有設定時，預設是 **Production**
- **launchSettings.json** 只在**自己的電腦**有效

後端：設定從哪裡來？



下面的蓋掉上面的。橘色的兩個用來放密碼。

appsettings.json

```
"Site": {  
  "Name": "Todo",  
  "FrontendUrl": "http://localhost:5174"  
},  
"Mail": { "Port": 587, "Password": "" }
```

appsettings.Staging.json

```
"Site": {  
  "Name": "Todo ( 測試環境 )",  
  "FrontendUrl": "http://localhost:5274"  
}
```

測試環境讀到的結果：

Site:Name = Todo (測試環境) ← 被蓋掉

Mail:Port = 587 ← 沒有被蓋掉

設定的名稱用 **:** 分層，例如 **Mail:Password**。環境變數改用兩個底線：**Mail__Password**。

密碼和 API key：不可以進 Git

環境	密碼放哪裡	誰設定
開發	User Secrets ：存在個人資料夾，不在專案裡	每個開發人員自己設定
測試、正式	環境變數 ，或雲端的密碼庫（例如 Azure Key Vault）	負責部署的人，或 CI 系統

開發環境：User Secrets

```
# 新專案先做一次：在 .csproj 加上 UserSecretsId
dotnet user-secrets init

dotnet user-secrets set "Mail:Password" "dev-password"
dotnet user-secrets set "Weather:ApiKey" "dev-key"
dotnet user-secrets list
```

測試環境：環境變數 (PowerShell)

```
$env:Mail__Password = "..."
$env:Weather__ApiKey = "..."

dotnet run --launch-profile staging
```

- `appsettings.json` 裡的密碼留空白 (`"Password": ""`)。它只說明「有這個設定」
- **User Secrets** 只在開發環境載入。它沒有加密，只是不在專案資料夾裡
- 連線字串裡有密碼時，整條連線字串都當成密碼

前端：Vite 的 .env 檔案

檔案	什麼時候載入
.env	每一次
.env.development	npm run dev
.env.staging	npm run dev:staging
.env.production	npm run build
.env.*.local	只在自己的電腦。不進 Git

- **mode** 決定讀哪一個檔案
 - .env.[mode] 的值蓋掉 .env 的值
- 只有 **VITE_** 開頭的變數會進到瀏覽器
 - WEB_PORT 只有 vite.config.js 讀得到
- 程式用 **import.meta.env.VITE_...** 讀取
 - 只在 config.js 讀，其他檔案 import 它

.env

```
# 所有環境共用
VITE_APP_TITLE=設定檢視器
WEB_PORT=5174
```

.env.staging

```
VITE_ENV_NAME=測試環境
VITE_API_URL=http://localhost:5248
WEB_PORT=5274
```

src/config.js

```
export const config = {
  mode: import.meta.env.MODE,
  envName: import.meta.env.VITE_ENV_NAME,
  apiUrl: import.meta.env.VITE_API_URL,
};
```

測試環境讀到的結果：

標題來自 **.env**，其他來自 **.env.staging**

前端的設定是公開的



建置好的檔案：`dist/assets/index-....js` (節錄)

```
var Ja={mode:`staging`,envName:`測試環境`,
  appTitle:`設定檢視器`,
  apiUrl:`http://localhost:5248`};
```

- Vite 在 `build` 時把值寫進 JS
- 按 `F12` 就看得得到
- `.gitignore` 擋不住這件事

前端沒有地方可以藏密碼。

需要 API key 的事交給後端：前端呼叫自己的後端，後端再呼叫別人的 API。

前端的 `.env` 只放公開也沒關係的值：後端的網址、網站的名稱、功能的開關。

六條規則，一張對照表

1 不進 Git

密碼不寫在程式和設定檔。
用 `.gitignore` 排除 `*.local`。

2 每個環境不同

開發、測試、正式各用各的密碼。
一個外洩，其他的不受影響。

3 權限越小越好

程式用的帳號只給需要的權限。
不要用管理員的帳號。

4 不顯示、不記錄

不寫進 `log`，不回傳給前端。
要顯示時，只顯示遮罩。

5 外洩了就立刻換

進過 `Git` 的密碼就當成外洩。
刪掉 `commit` 沒有用：先換密碼。

6 少了就馬上失敗

啟動時就檢查必要的設定。
不要等到執行一半才出錯。

設定的種類	後端放哪裡	前端放哪裡
一般的設定（網址、名稱）	<code>appsettings.json</code>	<code>.env</code>
每個環境不同的值	<code>appsettings.{環境}.json</code>	<code>.env.[mode]</code>
密碼、API key	User Secrets 或環境變數	不可以放
只給自己的電腦用	User Secrets	<code>.env.[mode].local</code>

啟動站台：用指令切換環境

開發環境 (Development)

終端機 1 : 後端 → localhost:5148

```
cd labs/06-config/ConfigApi
# 密碼放在 User Secrets ( 只要設定一次 )
dotnet user-secrets set "Mail:Password" "abc"
dotnet user-secrets set "Weather:ApiKey" "abc"
dotnet run
```

終端機 2 : 前端 → localhost:5174 (綠色)

```
cd labs/06-config/config-web
npm install
npm run dev
```

測試環境 (Staging)

終端機 3 : 後端 → localhost:5248

```
cd labs/06-config/ConfigApi
# 密碼放在環境變數 ( : 換成兩個底線 )
$env:Mail__Password = "abc"
$env:Weather__ApiKey = "abc"
dotnet run --launch-profile staging
```

終端機 4 : 前端 → localhost:5274 (橘色)

```
cd labs/06-config/config-web
# npm install 做過一次就可以
npm run dev:staging
```

程式是**同一份**。指令不同，讀到的設定就不同。
前端和後端要用**同一個環境**。兩個環境可以同時執行，並排比較。

Lab 6：動手做

LAB 6

- 1 後端：`cd labs/06-config/ConfigApi`，執行 `dotnet run` 它會失敗。讀錯誤訊息：少了什麼？
- 2 `dotnet user-secrets set "Mail:Password" "abc"`
`Weather:ApiKey` 也要設定。再執行 `dotnet run`：成功了
- 3 前端：`cd labs/06-config/config-web`
執行 `npm install` 和 `npm run dev`，開啟 `localhost:5174`
- 4 看畫面：每個設定的值從哪一個來源來？
密碼顯示成什麼？按 **F12** 找得到密碼嗎？
- 5 換成測試環境：照上一頁右邊的指令，再開兩個終端機
開啟 `localhost:5274`：畫面變成橘色，哪些值不同了？

挑戰

在 **Site** 區段加一個設定 **MaxTodos**：開發環境是 **5**，測試環境是 **100**。讓檢視器顯示它和它的來源。

開發環境

設定檢視器

前端的設定

來源：.env 檔案・build 的時候就放進 JavaScript・任何人都看得到・不可以放密碼・

名稱	值
MODE	development
VITE_ENV_NAME	開發環境
VITE_APP_TITLE	設定檢視器
VITE_API_URL	http://localhost:5148
WEB_PORT	(讀不到：沒有 VITE_ 開頭)

後端的設定

來源：GET http://localhost:5148/api/settings・密碼只留在後端・前端只拿到遮罩・

後端的環境：Development

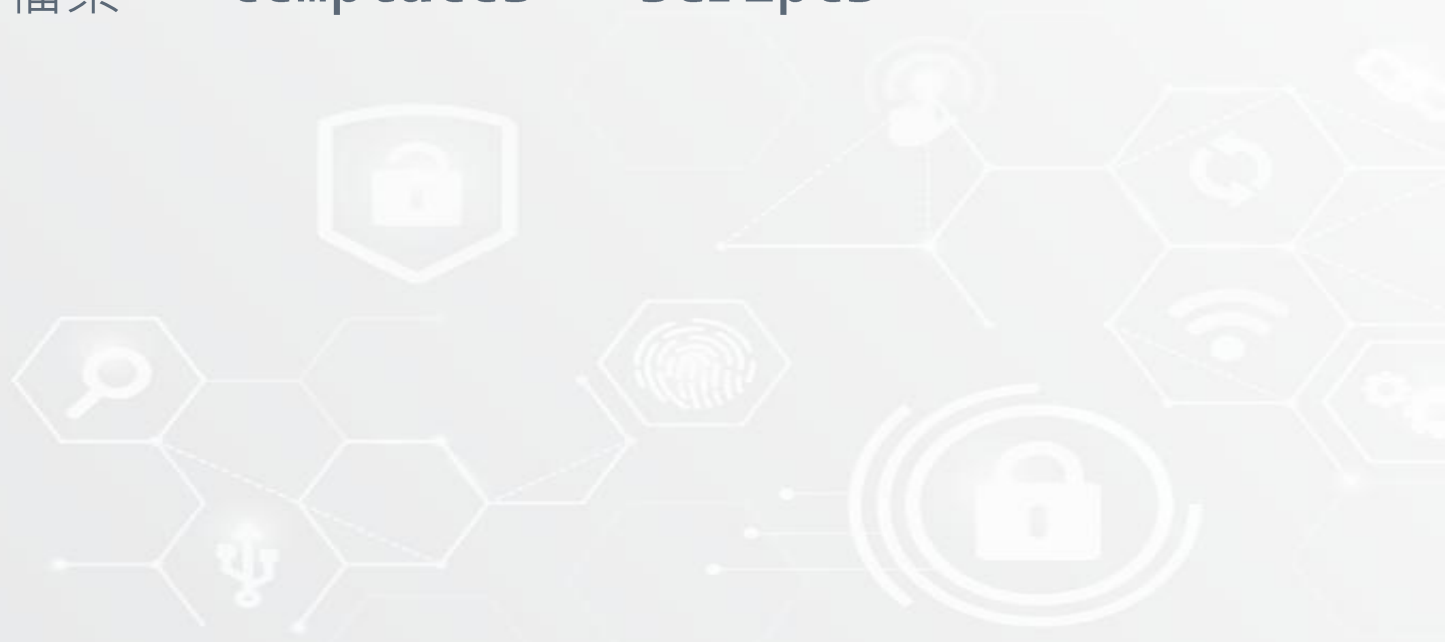
名稱	值	來源
Site:Name	Todo (開發環境)	appsettings.Development.json
Site:FrontendUrl	http://localhost:5174	appsettings.json
Mail:Host	localhost	appsettings.Development.json
Mail:Port	2525	appsettings.Development.json
Mail:Password		User Secrets
Weather:BaseUrl	https://api.weather.example	appsettings.json
Weather:ApiKey		User Secrets

綠色是開發環境。測試環境是橘色。

LAB 7

Skills

SKILL.md · 連結的檔案 · templates · scripts



Skill 是什麼？

Skill = 一個資料夾，裡面寫著「這件事要怎麼做」。Claude 需要時才讀它。

沒有 Skill

每次都要重新說明：
格式、步驟、規則。
每次的結果不一樣。

有 Skill

說明寫一次，存成檔案。
輸入 **/名稱** 就能使用。
結果的格式固定。

比喻：作業手冊

給新同事的**作業手冊**。
需要的時候才翻開，
不必每天背一次。

CLAUDE . md		Skill
什麼時候載入	每一次對話	需要時 才載入
適合放什麼	每次都要遵守的規則	特定工作的做法和流程

Skill 放在哪裡？怎麼啟動？

	Claude Code	Codex
個人	<code>~/.claude/skills/名稱/SKILL.md</code>	<code>~/.agents/skills/名稱/SKILL.md</code>
專案	<code>.claude/skills/名稱/SKILL.md</code>	<code>.agents/skills/名稱/SKILL.md</code>
子資料夾	<code>labs/07-claude-skills/.claude/skills/...</code>	目前資料夾的 <code>.agents/skills/...</code>
你啟動	輸入 <code>/名稱</code>	輸入 <code>\$</code> 選 Skill，或 <code>/skills</code>
自己啟動	依照 <code>description</code>	依照 <code>description</code>

你啟動：輸入 `/名稱`

例如 `/user-story` 加上清除已完成按鈕

`/名稱` 後面的文字會放進 `$ARGUMENTS`。

只讓人啟動：`disable-model-invocation: true`

工具自己啟動

它每次都會讀所有 Skill 的 `description`。

你說的話和它相符時，它自己載入這個 Skill。

所以 `description` 要寫清楚。

兩個工具的 SKILL.md 基本格式相同 (name、description)。同一個 Skill 可以放在兩個位置。

建立 Skill：一個資料夾，一個 SKILL.md

SKILL.md 的結構 (user-story 的開頭)

```
---
name: user-story
description: 把一句話的需求寫成 user story 和驗收條件。
  使用者說「寫 user story」、「把需求寫清楚」時使用。
argument-hint: [一句話的需求]
---
```

把下面的需求寫成 user story 和驗收條件。

需求：\$ARGUMENTS

步驟

1. 需求不清楚時，先問使用者問題。...

description 最重要。

寫清楚兩件事：**做什麼**，以及**什麼時候用**。

Claude 靠它決定要不要載入這個 Skill。

兩條 --- 的下面，是給 Claude 的說明。

- 1 建立資料夾 `.claude/skills/名稱/`
- 2 新增 `SKILL.md`：寫設定和說明
- 3 在 Claude Code 輸入 `/名稱` 試試看

設定	用途
name	指令的名稱
description	什麼時候使用
argument-hint	提示要輸入什麼
allowed-tools	可以直接用的工具

簡易範例：user-story (一個 md 檔)

.claude/skills/user-story/SKILL.md (節錄)

```
---
name: user-story
description: 把一句話的需求寫成 user story 和驗收條件。
  使用者說「寫 user story」、「把需求寫清楚」時使用。
argument-hint: [一句話的需求]
---

把下面的需求寫成 user story 和驗收條件。

需求：$ARGUMENTS

## 步驟
1. 需求不清楚時，先問使用者問題。一次最多問 3 個。不要自己猜。
2. 寫一個 user story：身為〈誰〉，我想要〈做什麼〉...
3. 寫 3 到 6 個驗收條件。每一個都用 Given、When、Then 的格式。
4. 驗收條件至少有一個「不正常的情況」。...
5. 列出「這次不做」的事，避免範圍變大。

## 輸出的格式 (省略)

## 規則
- 每個驗收條件只檢查一件事，而且可以寫成一個 E2E 測試。
- 不要寫怎麼實作。不要提到類別、資料表、函式的名稱。
```

在 Claude Code 輸入

/user-story 在 Todo App 加上「清除已完成」按鈕

Claude 的回答 (實際執行的結果，節錄)

```
## User story
身為 Todo App 的使用者，我想要按一個按鈕
就刪除所有已完成的項目，所以清單只留下
還沒做的事。

## 驗收條件
1. 一次清除所有已完成的項目
  - Given 清單有「買牛奶」(已完成) ...
  - When 使用者按「清除已完成」
  - Then 清單只剩下「寫作業」
3. 沒有已完成的項目時，看不到按鈕
5. 後端連不上時，顯示錯誤訊息

## 這次不做
- 復原被清除的項目 (undo)
```

一個 md 檔就是一個 Skill。\$ARGUMENTS 是 /user-story 後面的文字。說明越具體，結果越穩定。

進階範例： config-check 的結構

.claude/skills/config-check/	
config-check/	
SKILL.md	流程
rules.md	檢查規則的說明
fixes.md	修正方法
templates/report.md	報告的格式
scripts/scan-config.cs	掃描的腳本

部分	角色
SKILL.md	流程：先做什麼，再做什麼
連結的 md 檔	知識：需要時才讀
templates/	格式：每次的輸出都一樣
scripts/	確定的工作：交給程式做

執行的流程



橘色：程式做的事。藍色：Claude 做的事。

資料夾的名稱 (scripts、templates) 是習慣，不是規定。
重點：SKILL.md 要說明每個檔案的用途，以及什麼時候使用它。

連結其他檔案：需要時才讀

config-check/SKILL.md (節錄)

步驟

1. 執行掃描的腳本。
2. 腳本的結束代碼不是 0 時：告訴使用者，然後停止。
3. 讀 rules.md。
它說明每一條規則的意思和原因。
4. findings 不是空的時候，讀 fixes.md。
5. 讀 templates/report.md。
照這個格式寫報告。

- 用 Markdown 的連結：**[名稱](相對路徑)**
 - 路徑從 SKILL.md 所在的資料夾算起
- 寫清楚**什麼時候**要讀這個檔案
 - 例如：有問題時才讀 fixes.md
- **SKILL.md** 保持精簡：官方建議 **500 行以內**
 - 細節搬到連結的檔案。連結只要一層

三層載入

① description

每次對話都載入 (很短)



② SKILL.md

啟動這個 Skill 時載入



③ 連結的檔案

需要時才讀

沒有用到的檔案

不占對話的空間。

Skill 可以帶很多知識，
每次只讀需要的部分。

scripts/: .NET file-based app

scripts/scan-config.cs (簡化)

```
// 用法: dotnet run scan-config.cs -- <資料夾>
using System.Text.Json;

if (args.Length != 1 || !Directory.Exists(args[0]))
{
    Console.Error.WriteLine("找不到資料夾");
    return 1;
}

foreach (var file in ConfigFiles(args[0]))
{
    using var json = JsonDocument.Parse(File.ReadAllText(file));
    ScanJson(file, json.RootElement);
}

WriteReport(); // 輸出 JSON: 只有檔案和設定的名稱
return 0;

record Finding(string Rule, string File, string Key);
```

SKILL.md 裡的指令

```
dotnet run "${CLAUDE_SKILL_DIR}/scripts/scan-config.cs" -- "<要檢查的資料夾>"
```

腳本的輸出 (節錄)

```
{ "filesScanned": 4,
  "findings": [
    { "rule": "secret-in-appsettings",
      "file": "api/appsettings.json",
      "key": "Mail:Password" } ] }
```

- 一個 **.cs** 檔案就能執行 (.NET 10)
 - 不需要 **.csproj**，也不需要先建置
- 需要套件：**#:package 名稱@版本**
- 結果**每次都一樣**
- 密碼的內容**不輸出**，不進入對話

分工：程式做**確定**的事。
Claude 做需要**判斷**的事。

templates/：固定輸出的格式

templates/report.md (節錄)

設定檢查報告

- 檢查的資料夾：〈root〉
- 檢查的檔案數量：〈filesScanned〉
- 結果：〈沒有發現問題，或發現 N 個問題〉

發現的問題

#	嚴重程度	檔案	設定	問題	修正方法
---	------	----	----	----	------

請先做這件事

腳本找不到的問題

Claude 的報告 (實際執行的結果，節錄)

設定檢查報告

- 檢查的檔案數量：4
- 結果：發現 5 個問題 (高 4 個，中 1 個)

#	嚴重程度	檔案	設定
1	高	api/appsettings.json	Mail:Password
4	高	web/.env	VITE_WEATHER_API_KEY
5	中	web/.env	DEPLOY_TOKEN

請先做這件事

密碼只要進過 Git，就當成已經外洩。



- 範本是一個有[空格的文件](#)。〈 〉裡寫著要填什麼
- SKILL.md 說：「讀 templates/report.md，[照這個格式](#)寫報告」
- 好處：每次的報告[長得一樣](#)。容易比較，也容易交給別人
- 報告裡[沒有密碼的內容](#)：腳本沒有輸出它，SKILL.md 也不准 Claude 自己去找

Lab 7: 動手做

LAB 7

1 `cd labs/07-claude-skills`，再執行 `claude`
輸入 `/`：清單裡有 `user-story` 和 `config-check`

2 執行 `/user-story` 你的需求
看 Claude 寫的 `user story` 和驗收條件

3 執行 `/config-check sample-project`
報告列出幾個問題？密碼的內容有出現嗎？

4 執行 `/config-check ../06-config`
為什麼這個專案沒有問題？

5 修改 `user-story/SKILL.md` 的一條規則。
存檔後再執行一次：結果改變了嗎？

挑戰

建立自己的 `Skill`：把你最常重複說明的一件事，
寫成 `.claude/skills/名稱/SKILL.md`。

labs/07-claude-skills/

<code>.claude/skills/</code>	
<code>user-story/</code>	簡易
<code>config-check/</code>	進階
<code>sample-project/</code>	寫錯的設定

沒有 Claude Code 的人

讀兩個 `SKILL.md`，再直接執行腳本。
指令在這個資料夾的 `README.md`。

LAB 8

先搞懂需求，再實作

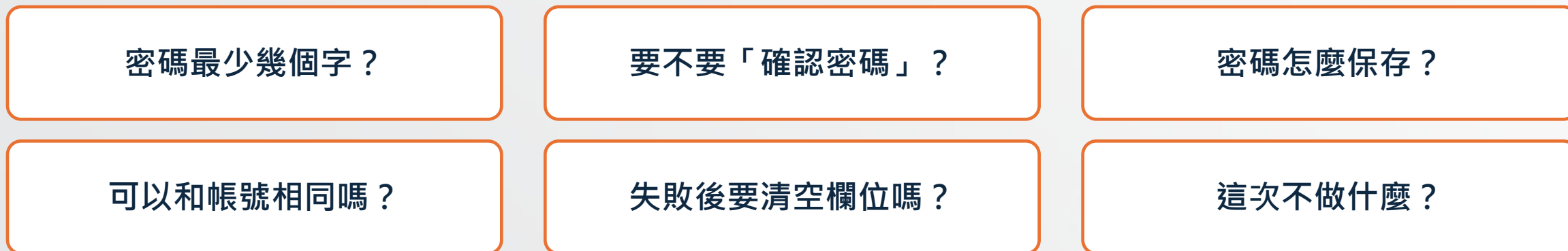
Spec-driven development · grill-me · 可被驗證的規格



一句話的需求：agent 會自己猜



這一句話**沒有回答**的問題：



你沒有說的事，agent 會**幫你決定**。
它選的答案，不一定和你想的相同。

SDD: 先寫規格，再寫程式

規格 (spec) = 一份可以被驗證的文件：寫著「做什麼」和「怎樣算完成」。



橘色：寫程式之前多做的兩步。

共同的依據

你和 agent 看同一份文件。
新的對話也可以讀它。

先改規格

需求改變時，先改 **spec.md**，
再請 agent 改程式。

工具

有 Spec Kit、Kiro 等工具。
今天只用一個 **md** 檔。

可被驗證的規格

程式和規格 100% 一樣嗎？沒有人可以保證。
LLM 會產生幻覺。所以規格要可以被驗證。

不能驗證

密碼要夠安全

錯誤時要提示使用者

密碼要安全地保存

可以驗證

密碼少於 8 個字：顯示「密碼最少 8 個字」

錯誤訊息顯示在那個欄位的下面

資料庫裡保存的值不是密碼本身

1 具體的值

8 個字，
不是「夠長」。

2 看得到的結果

畫面顯示什麼。
資料庫存了什麼。

3 通過或不通過

沒有「大概可以」。
人或測試都能檢查。

不要相信 agent 說的「做好了」。相信驗證的結果。

安裝 grill-me: 用 plugin

在終端機

```
claude plugins install mattpocock-skills
```

或是在 Claude Code 的對話裡

```
/plugin install mattpocock-skills
```

1 安裝 plugin : **mattpocock-skills**

2 啟動 **claude** , 輸入 **/grill**

3 清單裡有 **grill-me** : 安裝完成

plugin 在 Claude Code 的**官方 marketplace** 裡。
它有一整組 Skill。今天只用 **/grill-me**。

Skill (Lab 7)

自己寫。

放在 `.claude/skills` 資料夾。

Plugin (今天)

別人寫好的一整組 Skill。

一個指令就安裝。

grill-me 的 **SKILL.md** 只有一行：
呼叫另一個 Skill **grilling**。

grill-me: 讓 agent 先問你

在 Claude Code 輸入

/grill-me 我想在 ../05-tests 的會員註冊表單加上密碼欄位

agent 的第一輪問題 (實際執行的結果，節錄)

我讀了 05-tests 的程式碼。...系統沒有登入功能。

Q3 - 密碼的長度和字元規則：

- (a) 只限制長度：最少 8 個字，最多 64 個字
 - (b) 長度 + 組合規則 (大寫、小寫、數字)
 - (c) 你自己的規則
- 我建議 (a)。組合規則讓使用者選出可預測的密碼 (Password1!)，長度才是有效的保護。

Q5 - 確認密碼欄位：使用者看不到自己打的字。

- (a) 加第二個欄位「確認密碼」，兩個欄位要相同
 - (b) 加「顯示密碼」的切換按鈕
- 我建議 (a)，而且只在前端比對。

你的回答

Q3 : (a) 。 Q5 : (a) 。
Q2 : 我不懂，請解釋。

- 一輪問完現在可以問的問題
- 每一題有建議的答案
- 事實：agent 自己查
- 決定：是你的
- 沒有問題了，訪談才結束

不要每一題都回答「同意」。 看不懂：「請解釋」。 時間不夠：「其餘用你的建議」。

spec.md: 可被驗證的規格

訪談結束後輸入

把結論寫成 `spec.md`。不要實作。

`spec.md` (實際執行的結果, 節錄)

規格: 會員註冊表單加上密碼欄位

1. 目的

現在的註冊表單...會員沒有密碼。

2. 規則

條件 (符合就是錯誤)	錯誤訊息
密碼少於 8 個字	密碼最少 8 個字
密碼和帳號相同	密碼不可以和帳號相同

3. 驗收條件

AC-5 密碼太短

- Given 會員填了正確的身分證字號和帳號
- When 會員填入 7 個字的密碼 `abc1234`, 並按「註冊」
- Then 密碼欄位的下面顯示「密碼最少 8 個字」

4. 這次不做

- 登入
- 修改密碼

目的

為什麼做這件事

規則

條件和錯誤訊息

驗收條件

Given、When、Then

這次不做

避免範圍變大

測試

Lab 5

每一條驗收條件可以寫成一個測試。

agent 會出錯。

驗收條件全部通過, 才算完成。

Lab 8: 動手做

LAB 8

1 安裝 plugin :
`claude plugins install mattpocock-skills`

2 `cd labs/08-spec-first` , 再執行 `claude`
輸入 `/grill` : 清單裡有 `grill-me`

3 執行 `/grill-me` 我想在 `../05-tests` 的
會員註冊表單加上密碼欄位

4 回答第一輪的問題。至少一題自己決定。
看不懂的題目：「請解釋」

5 輸入「其餘用你的建議。把結論寫成 `spec.md`」
至少 3 條驗收條件，每一條都可以驗證

labs/08-spec-first/

README.md
spec.example.md

步驟
範例的結果

你的規格會和範例不同。
這是正常的：決定是你的。
但是每一條都要可以驗證。

挑戰

請 agent 依照 `spec.md` 實作，並加上測試。
再執行 Lab 5 的測試。

今天學到什麼

1 純前端

只用瀏覽器就能執行。
資料存在 `localStorage`。

2 Git 與 GitHub

`commit` 存檔，`push` 上傳。
`Pages` 把網頁放上網路。

3 ASP.NET Core

.NET 的版本和實作。
專案範本的結構。

4 MVC

伺服器用 `Razor`
組合 `HTML`。

5 Web API + Vue

後端回傳 `JSON`。
前端組合畫面。

6 測試

單元測試：規則的每種情況。
E2E：重要的流程。

7 設定管理

程式一份，設定多份。
密碼不進 `Git`。

8 Skills

把做法寫成檔案，
讓 `Claude` 重複使用。

9 先寫規格

讓 `agent` 先問你。
規格要可以被驗證。

下一步

延伸閱讀

- **API** 版加上「清除已完成」
 - 先用 `/grill-me` 弄清楚需求，再實作
- 加上「編輯標題」的功能
- **EF Core Migrations**：資料表改版
- 加上登入，每個人有自己的清單
- 把網站部署到雲端：設定用環境變數

學習資源

- **ASP.NET Core** 官方文件
 - learn.microsoft.com/zh-tw/aspnet/core
- **Vue 3** 教學
 - zh-hk.vuejs.org/tutorial
- **Pro Git** (中文版)
 - git-scm.com/book/zh-tw
- **Playwright for .NET**
 - playwright.dev/dotnet
- **Claude Code** 文件
 - code.claude.com/docs

Q & A