

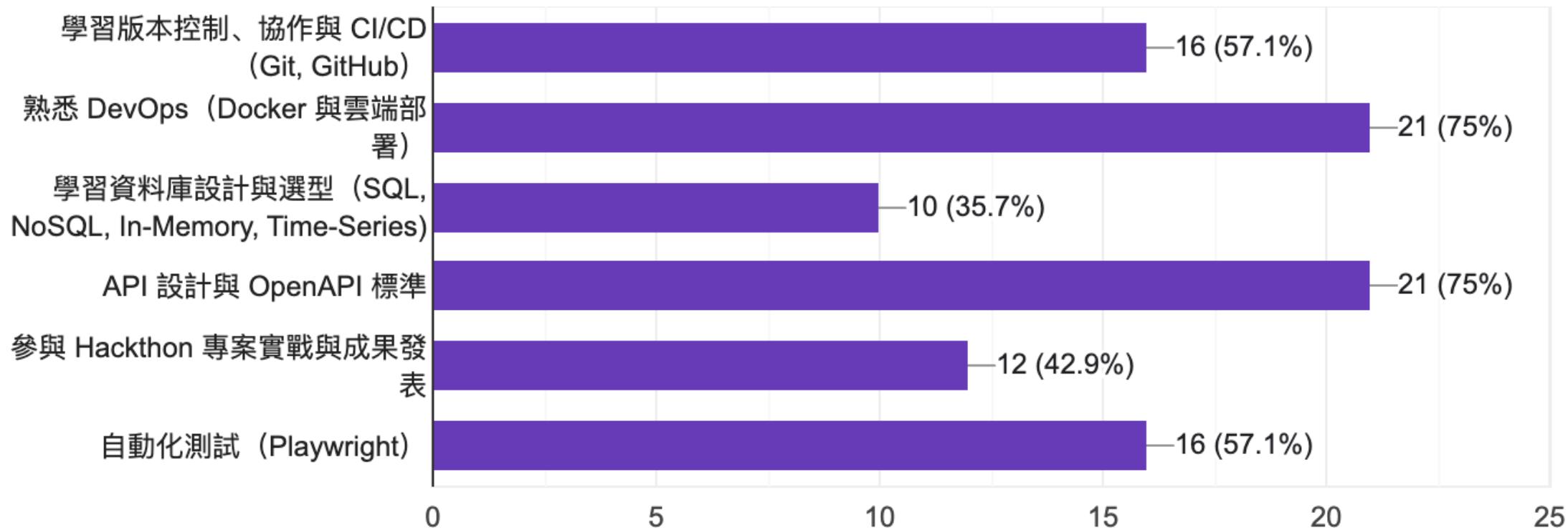
# 系統設計與分析 SAD 113-2

第 12 週課程：Todo App、Git、GitHub、Docker、Unit Tests、E2E Tests、CI/CD

助教：葉又銘、顧寬証，教授：盧信銘

# 課程目標

根據學期初的表單回饋，我們將著重於軟體開發流程、系統設計、DevOps 概念。



## Week 1 (今天)

- Git 版本控制、Docker 容器化、自動化測試 與 CI/CD 的基礎。
- 安裝與實作 Todo App，理解 DevOps 核心概念。

## Week 2

- Docker Compose：多容器管理，自動化前後端資料庫 CI/CD。
- 雲端原生架構：使用工作站部署實作。
- 帶大家操作管理虛擬機伺服器，假裝是個 IT 工程師。

## Week 3

- 資料庫選型：介紹各種不同的資料庫，帶大家理解技術選型與架構設計的概念
- OpenAPI：API 文件自動化生成，現代化 API 開發流程與 Best Practices
- 實作基本的 API 文件，並使用 Postman 測試 API

## Week 12 課程大綱

- 環境準備：安裝必要工具與軟體
- 從以 Todo app 為例，帶大家透過實際操作 Git、GitHub、Docker、Playwright、CI/CD 等工具理解 DevOps 的流程與概念

你可以到課程內容下載這份簡報，你也可以順便打開我們附上的教學文件幫助你跟上腳步

# 環境準備

## 必要工具

1. 電腦：Windows、Mac 或 Linux
2. 任一開發工具
3. 任一瀏覽器
4. GitHub 帳號
5. Windows 用戶建議：安裝 WSL 2 (Windows Subsystem for Linux 2)
6. Node.js：下載並安裝 [Node.js](#)，通常會順便安裝 npm

先給大家一點時間安裝！

# Node.js?

Node.js 是一個開源的 JavaScript 「執行環境」，讓我們可以在伺服器端（Server Side）運行 JavaScript 程式碼。

過去我們只能在瀏覽器上運行 JavaScript，而 Node.js 就是讓我們可以「執行」 JavaScript 的工具，就好像是 Python 的 `python` 指令一樣。

簡單來說，Node.js 讓我們可以使用 JavaScript 開發伺服器端的應用程式！

## npm?

NPM (Node Package Manager) 是 Node.js 的套件管理工具，讓我們可以輕鬆地安裝、更新和管理 JavaScript 套件。

## pnpm?

pnpm 是一款更高效、現代化的 Node.js 套件管理工具，本課程將使用它管理套件 (packages、dependencies) 。

總之就是現在的專案幾乎都是用 pnpm 來管理套件了，基本上指令幾乎跟 npm 一模一樣，使用 pnpm 會更快、更省空間。我們先來安裝吧！

## pnpm 安裝指令

```
npm install -g pnpm
```

## 驗證安裝

```
pnpm -v
```

## 為什麼選 pnpm ?

- 高效：安裝速度快，節省磁碟空間。
- 兼容性：支援 npm 指令，易於上手。
- 現代化：適合 monorepo 等架構。（等一下會介紹！）
- 甚至有些開源專案已經不再支援 npm 了...

## 再給大家一些安裝時間

請大家現在安裝教學文件上說明的按照指示，安裝必要工具。

# Git



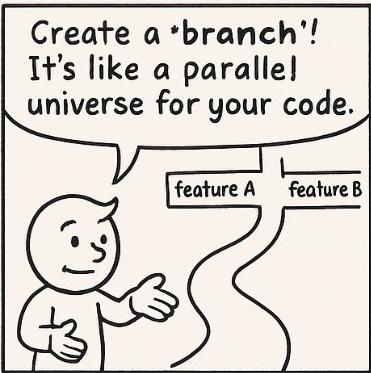
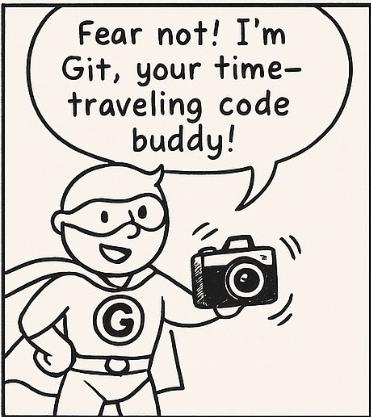
# Git 版本控制

## 為什麼需要 Git ?

Git 是現代開發的核心工具，幫助我們：

- **版本管理**：追蹤程式碼變更，隨時回溯。
- **團隊協作**：多人同時開發不衝突。
- **分支開發**：獨立開發新功能或修復 Bug。

## GIT STARTED WITH GIT!



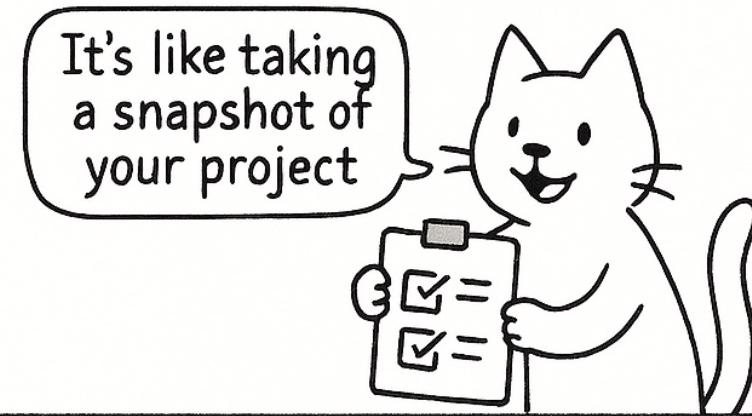
## Git 基本概念

- **Repository** (儲存庫) : 程式碼與歷史記錄的存放處。
- **Commit** (提交) : 記錄程式碼變更的快照 (snapshot) 。
- **Branch** (分支) : 獨立開發路徑。
- **Merge** (合併) : 整合不同分支的變更。

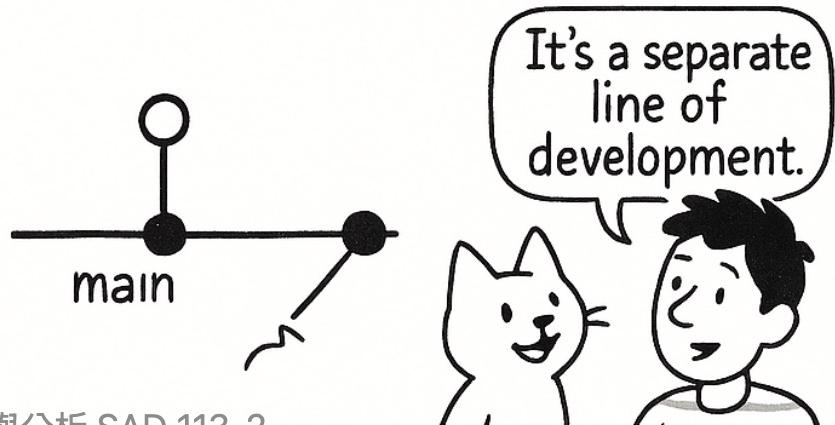
A **\*repository** is a collection of files.



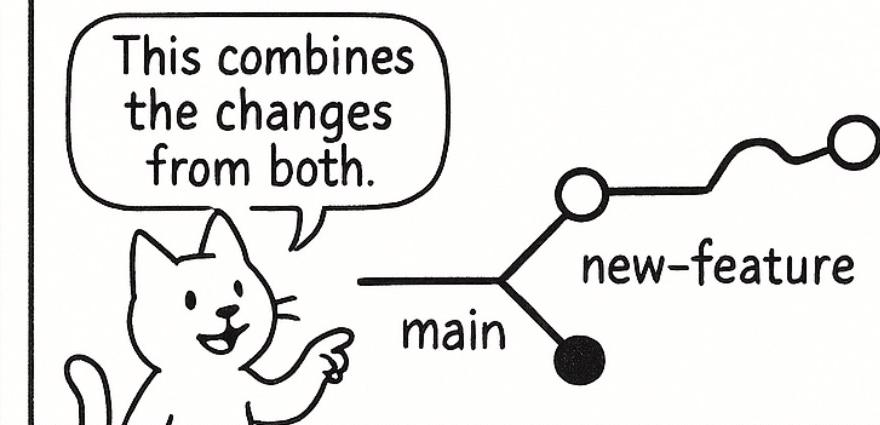
A **commit** saves your changes to the repository.



A **\*branch** lets you explore new ideas.



You can merge a branch back into the main line.



# Git 運作原理

Git 採用 快照 (snapshot) 機制管理版本，而非傳統的差異 (diff) 儲存。

## 核心機制

### 1. 快照記錄：

- 每次 Commit，Git 記錄所有檔案的完整狀態。
- 未變更的檔案只存指標，節省空間。

### 2. 物件模型：

- **Blob**：檔案內容。
- **Tree**：目錄結構。
- **Commit**：版本記錄（包含作者、時間等）。

### 3. SHA-1 雜湊值：

- 每個物件有唯一識別碼 (ID)

## 比喻

想像 Git 是一台「智慧型時光相機」：

- 按下快門（Commit）記錄整個場景。
- 未變動部分重用舊照片，節省空間。
- 相簿（`.git` 目錄）用指紋（SHA-1）管理照片。

`.git` 是在你的專案裡的隱藏資料夾，裡面存放著所有的版本控制資訊。

你可以用 `ls -a` 指令來查看這個資料夾，裡面有很多 Git 自己的檔案與資料夾。

# 安裝與設定 Git

## 安裝

- Windows : 下載 [Git](#)，包含 Git Bash。
- Mac : `brew install git`
- Linux : `sudo apt install git`
- 驗證 : `git --version`

## 基本設定

```
git config --global user.name "你的名稱"  
git config --global user.email "你的Email"
```

Git 會使用這些資訊來標記你的提交紀錄，讓其他人知道這些變更是由誰做的。

檢查 : `git config --list`

# GitHub



# GitHub 簡介

GitHub 是一個基於 Git 的雲端儲存庫平台，提供版本控制、協作開發和社群功能。

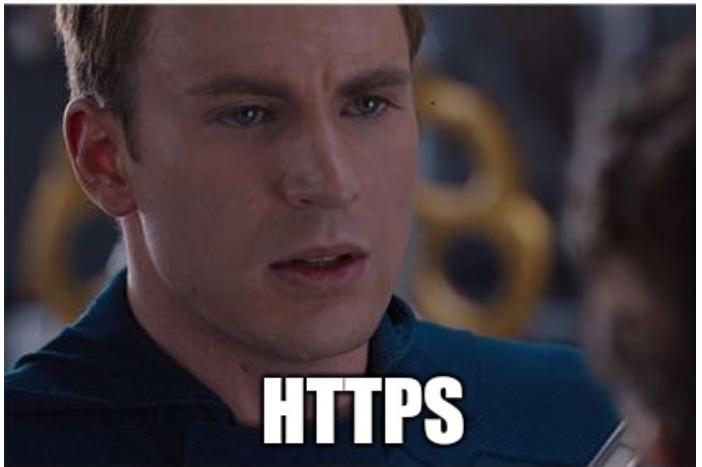
- **版本控制**：儲存與管理程式碼歷史。
- **協作開發**：多人同時開發，透過 Pull Request 進行程式碼審查。
- **社群功能**：問題 (Issues) 、Wiki、專案管理等。
- **開源專案**：支援開源專案，促進社群貢獻。
- **CI/CD**：整合 GitHub Actions，自動化測試與部署。（等一下會介紹！）

# SSH 設定

## 為什麼使用 SSH ?

讓 GitHub 與本地端安全連線，避免每次推送都要輸入帳號密碼（現在 GitHub 也不支援直接輸入密碼登入了），尤其是當你使用雲端機器時。

通常在雲端機器上不會使用 HTTPS 連線，因為這樣會需要每次都輸入帳號密碼，甚至 GitHub 還要求你使用 Token 來登入，這樣會更麻煩。



**HTTPS**



**SSH**



imgflip.com

# SSH 設定步驟

## 1. 生成金鑰：

```
ssh-keygen -t rsa -b 4096 -C "你的Email"
```

## 2. 添加公鑰至 GitHub：

- 查看公鑰：`cat ~/.ssh/id_rsa.pub`
- 在 [GitHub SSH 設定](#) 添加。

## 3. 測試連線：

```
ssh -T git@github.com
```

`ssh-keygen` 會生成兩個檔案，`id_rsa` 是私鑰，`id_rsa.pub` 是公鑰。私鑰要保密，公鑰可以分享給 GitHub。私鑰會用來加密你的連線，而公鑰則是用來讓 GitHub 驗證你身份。

## 實作時間

設定完記得截圖作為今天點名的證明！

## Git Flow：分支管理實踐

Git Flow 是一種 Git Branching Model（分支模型），是十幾年前提出的 Git 協作流程。

現在，根據不同的專案需求，Git Flow 也有很多變種，像是 GitHub Flow、GitLab Flow 等等。

### 分支模型

- **Main**：正式發布版本。（永遠是可部署狀態）
- **Develop**：開發中的程式碼。（整合所有功能）
- **Feature**：新功能開發分支。（從 Develop 分出）
- **Release**：準備發布的版本。（從 Develop 分出，用於測試與修復）
- **Hotfix**：緊急修復分支。（從 Main 分出，修復線上 Bug）



**NOOOOO!!! YOU CANT JUST PUSH  
TO MASTER!! YOU MUST PUSH TO A  
FEATURE BRANCH AND CREATE A  
PULL REQUEST! YOU ARE  
DISTORTING THE GIT FLOW!!!1**



**haha**  
`git push origin master --force`  
**go brrrr**

## 各分支詳解

- **Main (Master):**
  - 代表 穩穩定、已發布 的版本。
  - 只接受來自 `Release` 或 `Hotfix` 分支的合併。
  - 嚴禁 直接在此分支開發。
- **Develop:**
  - 整合 所有已完成的功能 ( `Feature` )。
  - 是 `Feature` 和 `Release` 分支的基礎。
  - 代表 下一個版本 的開發狀態。

## 各分支詳解 (續)

- Feature:

- 用於開發 新功能 。
- 必須 從 Develop 分支出來 。
- 完成後 必須 合併回 Develop 。
- 命名建議： feature/ 功能名稱 (e.g., feature/user-login) 。
  - | 但其實現在很多開源專案都不會限制命名規則了～

- Release:

- 用於 準備發布 新版本 。
- 從 Develop 分支出來 。
- 在此分支進行 測試、Bug 修復、版本號更新 。

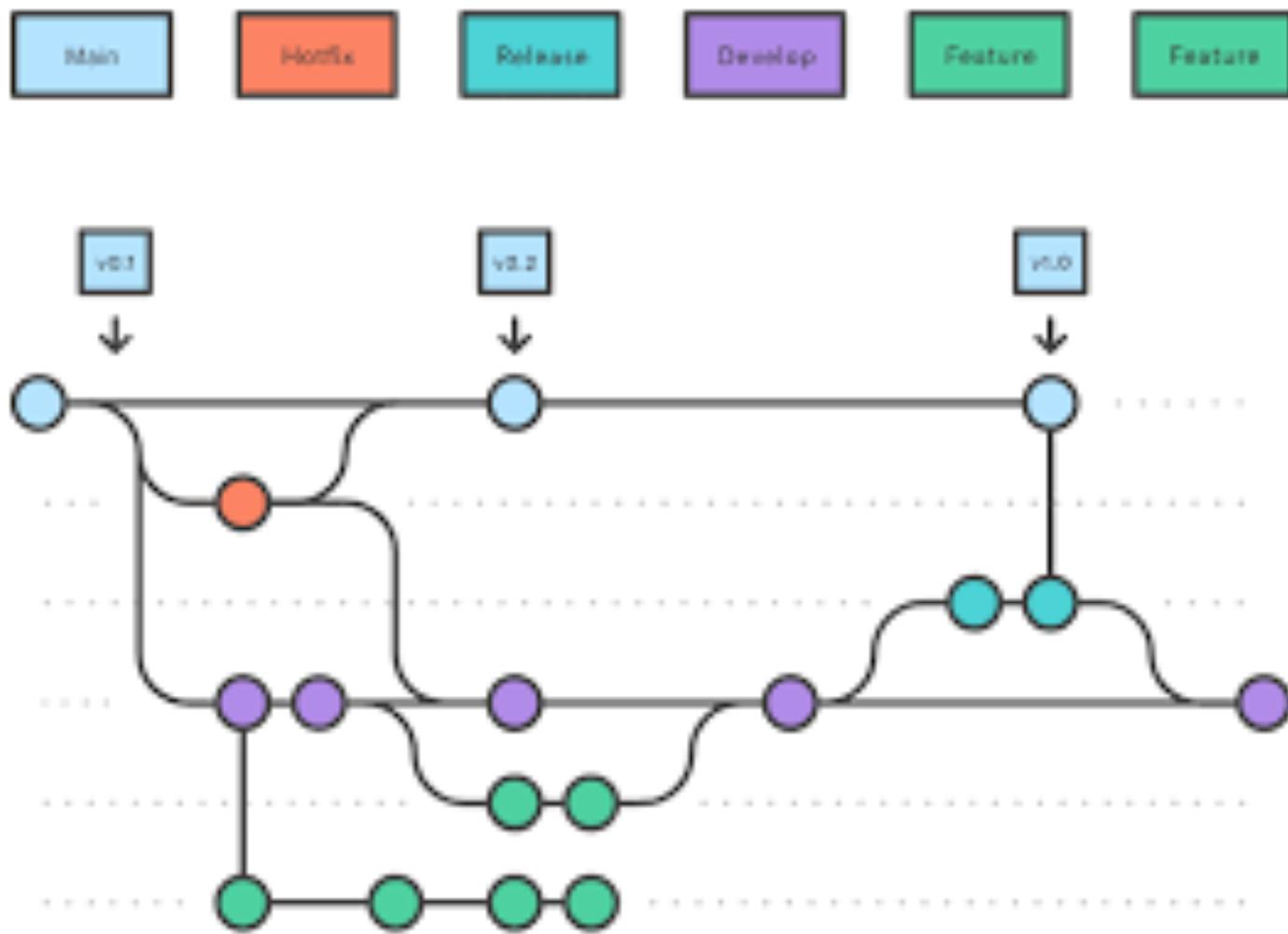
系統設計與分析 SAD 完成後 同時合併 回 Main (標記版本) 和 Develop (同步修復) 。

- **Hotfix:**

- 用於緊急修復 Main 分支上的 Bug。
- 必須從 Main 分支出來。
- 完成後同時合併回 Main 和 Develop。

以上是 Git Flow 的基本概念，實際上有些團隊會根據需求調整流程，但大致上都是這樣的運作方式就是了。

## 分支圖示



## Git Flow 協作流程

1. 開始新功能：從 `Develop` 建立 `feature/新功能` 分支。
2. 開發與提交：在 `feature` 分支上開發，頻繁提交變更。
3. 完成功能：將 `feature` 分支合併回 `Develop`。
4. 準備發布：當 `Develop` 累積足夠功能後，從 `Develop` 建立 `release/版本號` 分支。
5. 測試與修復：在 `release` 分支上進行測試，只做 Bug 修復。
6. 正式發布：
  - 將 `release` 分支合併到 `Main`，並打上版本標籤 (Tag)。
  - 將 `release` 分支合併回 `Develop`，確保修復也納入開發主線。
7. 緊急修復：若 `Main` 出現 Bug，從 `Main` 建立 `hotfix/問題描述` 分支，修復後合併回 `Main` 和 `Develop`。

## Git Flow 實戰場景：新增待辦事項截止日期

### 角色

- 小明：新加入的開發者
- Senior Lin：資深開發者，負責 Code Review
- PM Chen：專案經理，提出需求

### 需求

PM Chen 希望在 Todo App 中為每個待辦事項增加「截止日期」功能。

## 場景步驟 1：準備開發

1. **PM Chen**：在專案管理工具 (如 Jira, GitHub Issues) 開立新任務：「新增待辦事項截止日期」。
2. **小明**：接下任務，先確保本地 `develop` 分支是最新狀態。

```
git checkout develop  
git pull origin develop
```

3. **小明**：從 `develop` 分支建立新的 `feature` 分支。

```
git checkout -b feature/add-due-date develop  
# 分支名稱清晰表達功能目的
```

## 場景步驟 2：開發與提交

1. 小明：在 `feature/add-due-date` 分支上開發新功能（修改程式碼、加入 UI 元素等）。
2. 小明：開發過程中，進行多次小步提交 (Commit)。

```
# 修改檔案 ...
git add .
git commit -m "feat: add due date input field"
# 繼續修改檔案 ...
git add .
git commit -m "feat: implement due date saving logic"
```

3. 小明：功能初步完成後，將本地分支推送到遠端儲存庫 (GitHub)。

```
git push origin feature/add-due-date
```

## 場景步驟 3：建立 Pull Request (PR)

1. 小明：前往 GitHub 儲存庫頁面，針對 `feature/add-due-date` 分支，點擊「Create Pull Request」。
2. 設定 PR：
  - **Base Branch** : `develop` (要合併到的目標分支)
  - **Compare Branch** : `feature/add-due-date` (包含新功能的分支)
  - **標題**：清晰描述 PR 的目的 (e.g., "Feat: Add Due Date Feature")
  - **描述**：說明變更內容、如何測試、關聯的任務編號等。
  - **Reviewers**：指派 **Senior Lin** 進行 Code Review。

## 場景步驟 4：Code Review 與合併

1. Senior Lin：收到 PR 通知，檢視小明的程式碼變更。
2. 討論與修改 (可能發生)：
  - Senior Lin 可能會提出建議或要求修改，在 PR 頁面留言。
  - 小明 根據回饋，在本地 `feature/add-due-date` 分支修改程式碼，再次 commit 並 push。PR 會自動更新。
3. 批准 PR：Senior Lin 對程式碼滿意後，點擊「Approve」。
4. 合併 PR：
  - Senior Lin (或 小明，取決於團隊權限設定) 點擊「Merge Pull Request」。
5. 刪除分支 (可選)：合併後，可以安全地刪除遠端和本地的 `feature/add-due-date` 分支。

## 場景總結

- **PM Chen**：追蹤任務狀態
- **小明**：透過 Git Flow 貢獻了新功能
- **Senior Lin**：確保了程式碼品質

這個流程確保了新功能在獨立分支開發，經過審查後才合併，降低了直接修改主開發線的風險，也讓進度追蹤變得清楚。

# 試試看吧！

(詳細請參考教學文件)

1. Fork Todo App 專案：<https://github.com/113-2-SAD/SADo>
2. Clone 你的專案到本地端：`git clone git://<your-github-username>/SADo.git`
3. 從 `develop` 分支建立 `feature/your-feature-name` 分支
4. 隨意修改一行程式碼
5. 使用 `git add`、`git commit` 提交變更
6. Push 到 GitHub
7. 建立 Pull Request，並指派給自己
8. 等待自己 Code Review
9. 合併 PR，刪除分支

# Docker



# Docker 基礎與應用

## 為什麼用 Docker ?

解決開發痛點：

- 環境不一致：「我的電腦可以跑，別的不行。」
- 部署複雜：依賴管理繁瑣。
- 資源浪費：虛擬機過於笨重。

## Docker 簡介

Docker 是 容器化技術，將應用程式與依賴（Packages）打包成 映像檔（Image），在任何環境以 容器（Container）執行。

## Docker vs. 虛擬機

- VM：包含完整 OS，資源消耗大。
- Docker：共享主機 Kernel，輕量快速。

# Docker 運作原理

基於 Linux Namespace 與 Cgroups：

## 1. Namespaces：

- 隔離網路、檔案系統、進程等。
- 類型：PID、NET、MNT 等。

## 2. Cgroups：

- 限制 CPU、記憶體等資源。
- 確保容器不影響主機。

## Image 與 Container

- **Image**：應用程式模板，包含程式碼、依賴、環境設定。
- **Container**：Image 的執行實例，獨立運行的環境。

## 比喻

- Namespaces 是「隔離牢房」，Cgroups 是「資源管理員」。
- Docker 是「監獄」，確保每個應用安全運行。
- Image 是「囚犯制服」，Container 是「囚犯」。
- 這裡不是要同學們去理解這些技術細節，只是讓大家知道 Docker 其實也不是什麼黑魔法！

# 安裝與實作 Docker

## 安裝

- Windows/Mac：下載 [Docker Desktop](#)。
- Linux：

```
sudo apt install docker.io  
sudo systemctl enable docker
```

- 測試：`docker run hello-world`

這會下載一個叫做 `hello-world` 的 Docker Image，然後使用它來啟動一個 Container，這個 Container 會顯示一個歡迎訊息，告訴你 Docker 已經安裝成功了。

## Docker 基本指令與 Dockerfile

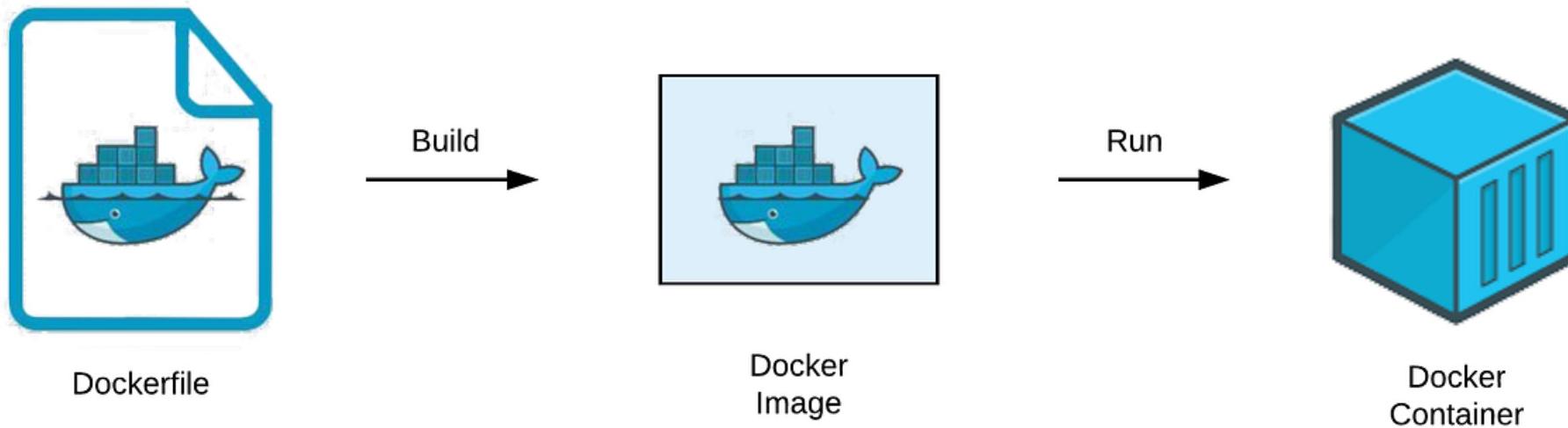
- **docker pull**：下載映像檔。
- **docker run**：啟動容器。
- **docker build**：建置映像檔。

## Dockerfile

描述映像檔的建置過程，包含基礎映像（Base Image）、安裝依賴、執行指令等。

Dockerfile 是一個純文字檔案，裡面包含了一系列的指令，這些指令告訴 Docker 如何建置這個映像檔。例如當你在打包一個專案時，你可以在 Dockerfile 裡面指定要使用哪個基礎映像、要安裝哪些套件、要執行哪些指令等等。

## Dockerfile 概念圖示



## 使用 Docker 建置 Todo App

### 1. Dockerfile (前端) :

(請參考教學文件)

### 2. 建構與運行：

```
docker build -t todo-app-frontend .
docker run -d -p 3000:3000 todo-app-frontend
```

### 3. 測試：瀏覽器訪問 <http://localhost:3000>。

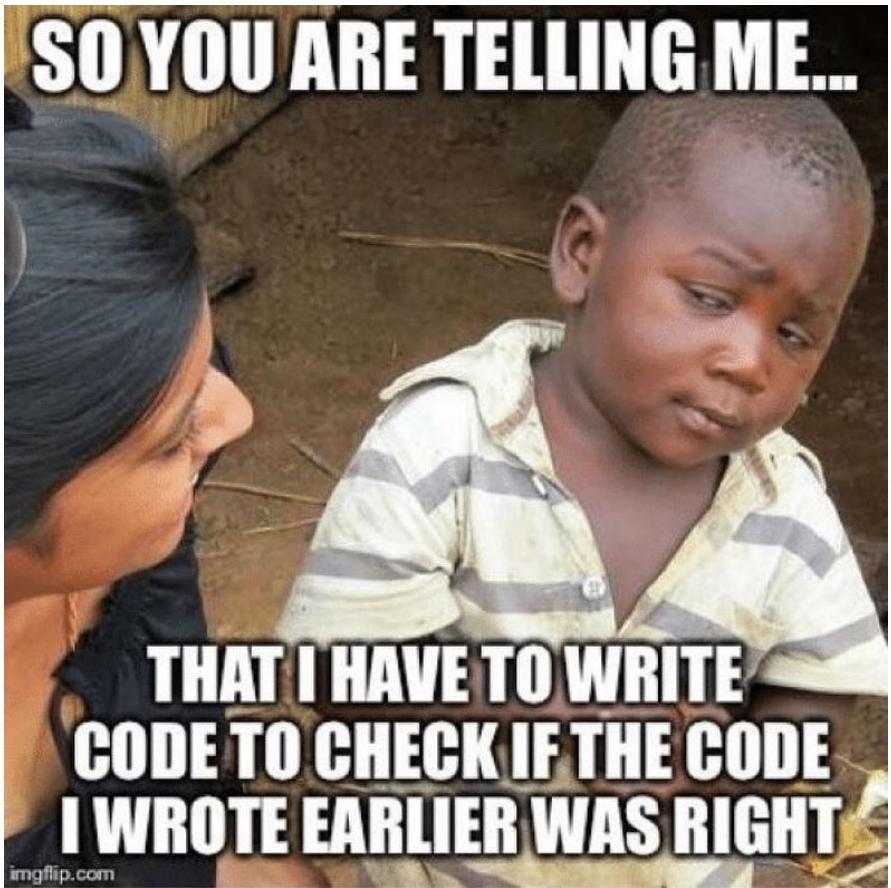
# 試試看吧！

(詳細請參考教學文件)

1. 運行 Todo App 的 Docker Frontend 映像檔
2. 運行 Todo App 的 Docker Backend 映像檔
3. 測試修改 Dockerfile 和運行指令，例如 port、環境變數等

# 軟體測試：為什麼需要測試？

- 品質保證：確保功能正常。
- 成本效益：早期發現 Bug，降低維護成本。
- 穩定性：提升使用者體驗，減少 Bug 回報。



Writing tests

## 測試類型

測試類型	說明
單元測試	測試單一功能或模組。
整合測試	測試多個模組之間的互動。
端對端測試 (E2E)	模擬使用者行為，測試整體系統。
性能測試	測試系統在高負載下的表現。
安全測試	測試系統的安全性與漏洞。
回歸測試	確保新功能不影響舊功能。

# 測試計畫 (Test Plan)

測試計畫是組織與規劃軟體測試活動的文件，確保測試過程完整且有效。

## 測試計畫的重要元素

- **測試範圍**：定義測試的功能與特性。
- **測試策略**：描述測試的方法與層次。
- **測試環境**：指定測試執行的軟硬體環境。
- **測試排程**：測試活動的時間安排。
- **測試資源**：人力、工具與其他必要資源。

好的測試計畫能讓團隊更系統性、有效率地執行，避免遺漏重要功能測試。

# 單元測試 (Unit Testing)

## 什麼是單元測試？

單元測試是測試程式碼最小單位（通常是函數或方法）的功能是否符合預期。

## 單元測試的特點

- **獨立性**：測試不依賴其他單元或外部系統。
- **自動化**：可重複執行，無需人工干預。
- **快速**：執行時間通常很短。
- **精確**：失敗時，能精確定位問題所在。

# 單元測試範例 (使用 Jest)

## 單元測試最佳實踐

- 測試獨立性：每個測試應獨立於其他測試。
- 測試覆蓋率：盡可能覆蓋所有程式碼路徑。
- 模擬依賴：使用 Mock 或 Stub 模擬外部依賴。
- 簡單明確：每個測試只測試一個行為或功能。

```
function add(a, b) {
  return a + b;
}
describe('數學函數測試', () => {
  test('add 函數正確相加兩個數字', () => {
    expect(add(1, 2)).toBe(3);
  });
});
```

# 從使用者故事到自動化測試

## 1. 使用者故事 (User Story)

使用者故事是從使用者角度描述軟體功能的簡短陳述。

例子：「作為使用者，我希望能夠登入系統，以便使用個人化功能。」

- 使用者故事描述了使用者期望的功能與目的
- 成為測試的基礎需求
- 幫助團隊理解實際使用情境

## 2. 測試計畫轉換

將使用者故事轉換成測試計畫 範例測試計畫 - Todo App :

針對「新增待辦事項」功能：

- 測試案例：驗證使用者能成功新增待辦事項
- 前置條件：Todo App 已啟動
- 步驟：輸入待辦事項、提交表單
- 預期結果：待辦事項出現在清單中

### 3. 手動測試腳本

將測試計畫轉換為明確的手動步驟：

1. 開啟瀏覽器，前往登入頁面
2. 在「使用者名稱」欄位輸入有效帳號
3. 在「密碼」欄位輸入正確密碼
4. 點擊「登入」按鈕
5. 確認系統跳轉至儀表板頁面
6. 確認頁面上有歡迎訊息與使用者名稱

## 手動測試的價值

軟體開發初期，撰寫嚴謹的測試計畫並以手動測試嚴格遵行，是相當有效的品質保證方法：

- **靈活適應**：能夠快速適應不斷變化的需求和界面
- **主觀評估**：能夠對使用者體驗、視覺呈現進行人為判斷
- **探索性強**：測試人員可以發現規格外的問題和邊界情況
- **成本效益**：初期不需花費時間建立自動化框架

不需要執著於測試的自動化，在開發初期，嚴謹的手動測試計畫同樣能夠有效確保品質。

# 端對端測試 (E2E Testing)

## E2E 測試的特點

- 完整性：測試整個應用流程，而非單一組件。
- 使用者視角：站在使用者角度測試系統，實際使用者如何與應用互動。
- 涵蓋範圍：測試系統所有組件的互動性。
- 執行時間：通常比單元測試或整合測試慢。

## 常用的 E2E 測試工具

- **Playwright**：支援多瀏覽器，功能強大的 E2E 測試框架。
- **Cypress**：專為現代網頁應用設計的測試工具。
- **Selenium**：廣泛使用的跨平台測試框架。

## 4. 將手動腳本轉換為自動化腳本（自動化測試腳本）

隨著專案規模成長，自動化測試的價值逐漸突顯：

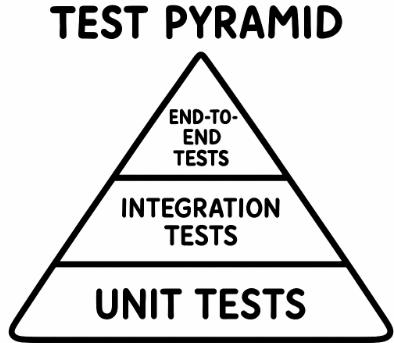
- 功能增加：手動測試所有功能變得耗時且難以執行
- 頻繁迭代：無法在每次程式碼變更後都進行完整手動測試
- 團隊效率：釋放測試人員處理更高價值的探索性測試

當軟體規模逐漸龐大，無法每次開發新功能或改動程式碼就安排人力手動測試每個功能，自動化測試成為必要投資。

## E2E 測試最佳實踐

- **選擇關鍵流程**：專注於測試用戶關鍵路徑（如登入、購買流程）。
- **測試數據管理**：使用獨立的測試數據，避免影響生產環境。
- **並行執行**：配置測試能夠並行運行，減少執行時間。

# 測試金字塔



測試金字塔是一種視覺化測試策略的模型，從下到上：

- 底層：單元測試 - 數量最多，執行最快，成本最低
- 中層：整合測試 - 測試組件間的互動
- 頂層：E2E 測試 - 數量最少，執行最慢，但最接近真實使用者體驗

良好的測試策略應在各層級間取得平衡，根據專案需求調整比例

# 單元測試實作：Jest 入門

## Jest 簡介

Jest 是由 Facebook 開發的 JavaScript 測試框架，專為現代 Web 開發設計：

- **零配置**：開箱即用，無需複雜設定
- **內建功能**：斷言、模擬、覆蓋率報告等
- **隔離環境**：每個測試在獨立環境執行，避免相互影響

## Jest 安裝與設定

```
# 安裝 Jest  
pnpm add -D jest
```

```
# 在 package.json 中設定測試指令  
# "scripts": {  
#   "test": "jest"  
# }
```

## Jest 基本測試結構

```
// 測試檔案命名通常為：功能名.test.ts(x)，引入待測試的函數
import { sum, multiply } from './math';
// 測試套件 (Test Suite)
describe('數學函數測試', () => {
  // 測試案例 (Test Case)
  test('加法函數能正確相加兩個數字', () => {
    // 使用 expect 與 matcher 進行斷言
    expect(sum(1, 2)).toBe(3);
  });
  test('乘法函數能正確相乘兩個數字', () => {
    expect(multiply(2, 3)).toBe(6);
  });
});
```

## Jest 常用斷言 (Matchers)

```
// 相等性測試
expect(value).toBe(exactValue);           // 嚴格相等 ( === )
expect(value).toEqual(object);             // 深度比較 (對物件有用)

// 真假值測試
expect(value).toBeTruthy();                // 轉換為 true
expect(value).toBeFalsy();                  // 轉換為 false

// 數字比較
expect(value).toBeGreaterThan(3);          // 大於
expect(value).toBeLessThanOrEqual(1);       // 小於等於

// 字串測試
expect(string).toMatch(/regex/);           // 符合正則表達式

// 陣列測試
expect(array).toContain('item');            // 包含項目

// 例外測試
expect(() => func()).toThrow();          // 拋出例外
```

## Jest 實作：Todo 函數測試

```
// todo.js
export function addTodo(todos, text) {
    return [...todos, { id: Date.now(), text, completed: false }];
}

export function toggleTodo(todos, id) {
    return todos.map(todo =>
        todo.id === id ? { ...todo, completed: !todo.completed } : todo
    );
}
```

## 續上頁

```
// todo.test.js
import { addTodo, toggleTodo } from './todo';
describe('Todo 功能測試', () => {
  test('能新增待辦事項', () => {
    const initialTodos = [];
    const newTodos = addTodo(initialTodos, '學習 Jest');
    expect(newTodos.length).toBe(1);
    expect(newTodos[0].text).toBe('學習 Jest');
    expect(newTodos[0].completed).toBe(false);
  });
  test('能切換待辦事項完成狀態', () => {
    const initialTodos = [{ id: 1, text: '學習測試', completed: false }];
    const updatedTodos = toggleTodo(initialTodos, 1);
    expect(updatedTodos[0].completed).toBe(true);
  });
});
```

## Jest 模擬 (Mocks)

用於隔離測試單元，模擬外部依賴：

```
// 模擬函數
const mockCallback = jest.fn();
mockCallback.mockReturnValue(42); // 設定回傳值
jest.mock('./math'); // 自動模擬整個模組
// 模擬 API 請求
global.fetch = jest.fn(() =>
  Promise.resolve({
    json: () => Promise.resolve({ data: 'mocked data' })
  })
);
// 測試 API 請求函數
test('fetchTodos 應正確取得資料', async () => {
  const data = await fetchTodos();
  expect(data).toEqual({ data: 'mocked data' });
  expect(fetch).toHaveBeenCalledTimes(1);
});
```

# 執行 Jest 測試

```
# 執行所有測試  
pnpm test  
# 執行特定測試檔案  
pnpm test -- todo.test.js  
# 監視模式 (變更時自動執行)  
pnpm test -- --watch  
# 生成覆蓋率報告  
pnpm test -- --coverage
```

## 自動化測試建議

1. 測試重要功能：優先測試核心業務邏輯
2. 測試錯誤情況：包含邊界值和例外處理
3. 保持測試獨立：避免測試間的相互依賴
4. 持續整合：配合 CI/CD 流程使用

# 現代軟體測試介紹：Playwright 入門

## Playwright 簡介

- 特點：支援多瀏覽器、E2E 測試、直觀 API。
- 優勢：開發社群、無頭模式（不用真的開瀏覽器）、直觀設計。

現在很多用 AI 操控瀏覽器的工具其實也是基於 Playwright，像是 Manus、Browser Use 等。

# 現在一起試試看吧！

記得把執行結果截圖當作今天的點名證明！

## 安裝 (詳細內容參考文件)

```
pnpm install -D @playwright/test  
pnpm dlx playwright install
```

## 測試範例

tests/todo.spec.js :

```
import { test, expect } from "@playwright/test";
test("新增待辦事項", async ({ page }) => {
    await page.goto("http://localhost:3000");
    await page.fill('input[name="title"]', "學習");
    await page.click('button[type="submit"]');
    expect(await page.innerText(".todo-item")).toContain("學習");
});
```

執行 : pnpm test (先確保剛剛使用 Docker 啟動了前端應用程式)

# DevOps



# DevOps 簡介

- DevOps：開發（Development）與運營（Operations）的結合。
- 目標：提升軟體交付速度與品質。
- 文化：強調團隊合作、持續改進與自動化。
- 工具：CI/CD、容器化、監控等。
- 流程：持續整合（CI）、持續交付（CD）、持續部署（CD）。
- 價值：縮短開發週期、提高產品質量、增強團隊協作。
- 實踐：自動化測試、持續交付、基礎設施即代碼（Infrastructure as Code）。

# CI/CD 實踐：GitHub Actions

## CI/CD 簡介

- CI（持續整合）：自動測試新程式碼。
- CD（持續部署）：測試通過後自動部署。
- DevOps 角色：提升交付速度與品質。

過去我們軟體交付速度慢，因此手動測試、部署並沒有什麼問題，但現在我們的軟體強調是快速迭代、敏捷開發，因此我們需要自動化測試與部署的流程，這樣才能快速地交付新的功能給使用者。

## CI/CD 流程

1. 開發者提交程式碼到 Git 儲存庫（例如 Push、Pull Request 等）。
2. 在合併前，CI 工具自動開始檢查程式碼，執行測試。
3. 測試通過後，CI 工具會通知開發者或直接合併程式碼。
4. 當程式碼合併到主分支後，CD 工具會自動將最新版本的應用程式部署到生產（Production）環境。
5. 部署完成後，CD 工具會通知相關人員，並提供部署狀態報告。
6. 若部署失敗，CD 工具會自動回滾到上個穩定版本，並通知相關人員。

## GitHub Actions

GitHub 提供的 CI/CD 工具：

1. 在專案新增 Workflow 檔案 ( `.github/workflows/ci.yml` ) :

```
name: CI
on: [push]
jobs: (詳細請參考文件)
```

2. 推送後自動執行，檢查結果於 GitHub Actions 頁面。

# 試試看吧！

(詳細請參考教學文件)

1. 在 Todo App 專案中新增 GitHub Actions Workflow
2. 設定 CI/CD 流程
3. 測試自動化測試與部署
4. 修改程式碼，觀察 CI/CD 流程

# 總結與下一步

## 本週回顧

- **Git**：版本控制與 Git Flow。
- **Docker**：容器化基礎與實作。
- **Playwright**：自動化測試。
- **CI/CD**：GitHub Actions 入門。

## 下一週

- **Docker Compose**：多容器管理。
- 雲端原生架構：進階部署實作。

# 感謝大家聆聽！

有任何問題或建議，歡迎隨時提出。

記得將實作結果截圖作為點名證明哦！