系統設計與分析 SAD 113-2

第 13 週課程：Docker Compose & 初探雲端原生部署

助教：葉又銘、顧寬証，教授：盧信銘

上週回顧 & 本週議程

上週 (Week 12) 重點

Git 版本控制與 GitHub 協作, Docker 基礎：Dockerfile, Image, Container, 自動化測試：單元測試 (Jest), E2E 測試 (Playwright), CI/CD 概念與 GitHub Actions 實作

本週 (Week 13) 議程

- **Docker Compose**：為什麼需要？如何使用？
- `docker-compose.yml` 檔案結構與常用指令
- **實作**：使用 Docker Compose 運行 Todo App (前後端)
- **雲端原生 (Cloud Native)** 簡介
- **實作**：將 Todo App 部署到工作站伺服器 (模擬雲端部署)
- 伺服器基本操作與應用程式管理

Git 常用指令複習

- 初始化專案（創建一個 .git 資料夾來管理專案 Repository）：
  `git init`
- 加入檔案（把本目錄檔案加入暫存區，代表要追蹤這些檔案）：
  `git add .`
- 提交變更（把暫存區的變更提交到本地 Repository）：
  `git commit -m "訊息"`
- 提交變更到遠端（把本地的變更推送到遠端 Repository，如 GitHub）：
  `git push origin 分支名`
- 下載遠端（把遠端的變更拉取到本地）：
  `git pull origin 分支名`
  `git fetch origin`（下載遠端的變更，但不合併）

- 查看狀態（如果有檔案變更或未追蹤的檔案）：
  `git status`
- 查看紀錄（查看 Commit 歷史）：
  `git log`
- 建立並切換分支：
  `git checkout -b feature/xxx`
- 切換分支：
  `git checkout feature/xxx`
- 合併分支（把指定的分支合併到當前分支）：
  `git merge feature/xxx`

Git 使用情境

小明是一位開發者，今天他要為 Todo App 新增「使用者註冊」功能。

1. **開始新任務**：
   - 首先，小明切換到 `develop` 分支並拉取最新程式碼，確保基礎是最新的。

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

   - 接著，他為新功能建立一個名為 `feature/user-registration` 的分支。

     ```
     git checkout -b feature/user-registration
     ```

Git 使用情境 (續)

2. **開發功能**：
   - 小明新增了 `auth.js` 檔案並修改了 `server.js` 來處理註冊邏輯。
   - 他想看看目前的變更狀態：

```
git status
# 會顯示 auth.js 是新檔案 (untracked), server.js 已修改
```

   - 他將這些變更加入暫存區：

```
git add auth.js server.js
# 或 git add . (加入所有變更)
```

   - 提交這次的進度：

```
git commit -m "feat: add user registration endpoint and logic"
```

Git 使用情境 (再續)

3. **繼續開發與推送**：
   - 小明又寫了一些前端註冊表單的程式碼，並再次 `git add .` 和 `git commit -m "feat: create registration form UI"`。
   - 他想看看最近的提交紀錄：

     ```
     git log -n 2 --oneline
     # 顯示最近兩筆 commit
     ```

   - 功能初步完成，他將 `feature/user-registration` 分支推送到遠端 GitHub，方便同事 Code Review 或備份。

     ```
     git push origin feature/user-registration
     ```

Git 使用情境 (完)

4. **同事的更新與合併 (假設情境)**：
    - 隔天，同事小華在 `develop` 分支上修復了一個 Bug 並已推送到遠端。
    - 小明需要將這些更新同步到自己的功能分支：

```
git checkout develop       # 切回 develop
git pull origin develop    # 更新 develop
git checkout feature/user-registration # 切回功能分支
git merge develop          # 將最新的 develop 合併進來
                           # (若有衝突，需手動解決)
```

    - 解決衝突後 (如果有的話)，小明再次提交並推送。

這個情境展示了分支、新增、提交、推送、拉取、合併等常用 Git 操作。

Docker 常用指令複習

- `docker run <image_name_or_id>`：運行容器
- `docker build -t <image_name> .`：使用 Dockerfile 建置映像檔
- `docker images`：列出所有映像檔
- `docker ps`：列出容器，`docker ps -a`：列出所有容器
- `docker logs <container_name_or_id>`：查看日誌
- `docker exec -it <container_name_or_id> sh`：進入容器
- `docker stop <container_name_or_id>`：停止容器
- `docker rm <container_name_or_id>`：移除容器
- `docker rmi <image_name_or_id>`：移除映像檔
- `docker pull <image_name_or_id>`：拉取映像檔

Docker 使用情境

小華是一位後端開發者，她正在開發一個新的 Node.js 服務，並希望使用 Docker 來打包和運行它，以確保環境一致性。

1. **尋找基礎映像檔**：
   - 小華知道她的服務需要 Node.js 環境，所以她先到 Docker Hub 查找官方的 Node.js 映像檔。
   - 她決定使用 `node:18-alpine` (一個輕量級的版本)。

```
docker pull node:18-alpine
# 下載映像檔到本地
docker images
# 確認 node:18-alpine 已在列表中
```

Docker 使用情境 (續)

2. **撰寫 Dockerfile**：
   - 小華在她的專案根目錄下建立了一個 `Dockerfile`：

```
# Dockerfile
FROM node:18-alpine
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 8080
CMD [ "node", "server.js" ]
```

   - 這個 Dockerfile 定義了如何建置她的應用程式映像檔。

Docker 使用情境 (再續)

3. **建置映像檔**：
    - 有了 `Dockerfile` 和應用程式碼 (假設 `server.js` 和 `package.json` 已準備好)，小華開始建置她的 Docker 映像檔。

```
docker build -t my-node-service:1.0 .
# -t my-node-service:1.0 給映像檔取名為 my-node-service 並標記版本為 1.0
# . 表示 Dockerfile 在當前目錄
docker images
# 確認 my-node-service:1.0 已成功建置
```

Docker 使用情境 (又續)

4. **運行容器**：
   - 映像檔建置完成後，小華嘗試在本機運行它。

```
docker run -d -p 8080:8080 --name myapp my-node-service:1.0
# -d: 背景執行
# -p 8080:8080: 將主機的 8080 port 映射到容器的 8080 port
# --name myapp: 給容器取一個名字叫 myapp
docker ps
# 查看正在運行的容器，應該能看到 myapp
```

   - 她打開瀏覽器或使用 curl 測試 `http://localhost:8080`，確認服務正常。

Docker 使用情境 (完)

5. **偵錯與管理**：
   - 如果服務沒有如預期運行，小華會查看容器的日誌：

```
docker logs myapp
```

   - 有時她需要進入容器內部檢查檔案或環境：

```
docker exec -it myapp sh
# 進入 myapp 容器的 shell 環境
```

   - 測試完畢後，她停止並移除容器：

```
docker stop myapp
docker rm myapp
```

這個情境展示了拉取映像檔、撰寫 Dockerfile、建置映像檔、運行容器以及基本的容器管理指令。

Docker Compose

為什麼需要 Docker Compose ?

管理多容器應用的挑戰

回想一下我們的 Todo App：

- 前端 (Frontend) 服務
- 後端 (Backend) 服務
- 資料庫 (Database) 服務

如果用 `docker run` 指令個別啟動：

- 需要手動管理多個容器的啟動順序，例如資料庫需要先啟動，後端才能啟動。
- 網路設定複雜 (例如：前端如何找到後端 API？)。
- 連接埠映射 (Port mapping) 容易混亂，有時候一台機器上開了十幾個服務，連接埠很容易撞在一起。
- 更新或重啟多個服務很麻煩，你需要一次輸入很多指令。
- 指令需要各種複製貼上。

> **想像一下，如果你的應用有十幾個微服務，手動管理會是一場災難！**

Docker Compose 是什麼？

Docker Compose 是一個用來 **定義和執行多容器 Docker 應用程式** 的工具。

- 使用一個 **YAML 檔案 (`docker-compose.yml`)** 來設定應用程式的所有服務 (services)。
- 只需要一個指令，就可以從設定檔中建立並啟動所有服務。

主要優點

- **簡化設定**：將複雜的多容器設定集中管理。
- **一致環境**：確保開發、測試、生產環境的一致性。
- **快速啟動/停止**：一鍵管理整個應用程式堆疊。
- **易於擴展**：方便地增加或修改服務。

> 現在 Docker Compose 已經是 Docker 的一部分，也進化到可以與 Docker Swarm、Stack 等更複雜的架構整合

`docker-compose.yml` 檔案結構

這是一個 YAML 格式的設定檔，通常放在專案的根目錄。

主要包含以下幾個部分：

- **`version`**：指定 Docker Compose 檔案格式的版本 (通常是 `'3.8'` 或類似)。
- **`services`**：定義應用程式中的各個服務 (容器)。
  - 每個服務可以有自己的 `build` (使用哪個 Dockerfile 建置映像檔)、`image` (直接指定映像檔)、`ports` (連接埠映射)、`volumes` (用於持久化資料，例如資料庫的資料不能只活在容器內)、`environment` (環境變數)、`depends_on` (誰先啟動，例如資料庫需要先啟動，後端才能啟動)、`restart` (重啟策略，這個服務掛了要不要自動重啟) 等設定。
- **`networks`**：服務之間要怎麼通訊。

`docker-compose.yml` 範例 (概念)

```yaml
# docker-compose.yml
version: '3.8' # 指定 Compose 檔案版本

services: # 定義所有服務
  frontend: # 前端服務名稱
    build:
      context: ./frontend # Dockerfile 所在路徑
      dockerfile: Dockerfile # 可選，預設為 Dockerfile
    ports:
      - "3000:3000"
    depends_on:
      - backend
    environment:
      - REACT_APP_API_URL=http://backend:8080/api
    # volumes:
      # - ./frontend/src:/app/src # 開發時掛載程式碼以實現熱重載
    networks:
      - app-network
```

```yaml
backend: # 後端服務名稱
  build: ./backend
  ports:
    - "8080:8080"
  volumes:
    - backend_data:/app/data # 範例：掛載持久化資料
  environment:
    - DATABASE_URL=postgres://user:password@db:5432/mydb
    - NODE_ENV=development
  networks:
    - app-network
    - db-network # 後端可以同時連接到應用網路和資料庫網路
  restart: unless-stopped # 設定重啟策略
```

```
  db: # 資料庫服務
    image: postgres:13
    ports:
      - "5432:5432" # 開發時可映射，生產環境通常不對外暴露
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: mydb
    volumes:
      - postgres_data:/var/lib/postgresql/data # 持久化資料庫數據
    networks:
      - db-network # 資料庫僅連接到資料庫網路
    restart: always

volumes: # 定義具名資料卷
  postgres_data:
  backend_data:

networks: # 定義自訂網路
  app-network:
    driver: bridge
  db-network:
    driver: bridge
```

> **提示**：開發環境與生產環境的 `docker-compose.yml` 可能會有所不同 (例如使用 `docker-compose.override.yml` 來擴展基礎配置)。

Docker Compose 常用指令

- **`docker-compose up`**: 建立並啟動所有服務。
  - `-d` (detached mode): 在背景執行。
  - `--build`: 在啟動前重新建置映像檔。
- **`docker-compose down`**: 停止並移除服務、網路、資料卷。

> 基本上你只要記住 `docker-compose up -d --build` 這個指令就夠了！

實作時間：Todo App with Docker Compose

目標

使用 `docker-compose.yml` 一次啟動 Todo App 的前端和後端服務。

跟著做

- 請依照教學文件指示，並跟隨助教的引導完成操作。
- **目標**：在本機成功使用 Docker Compose 運行前後端服務。
- **點名**：截圖 `docker-compose ps` 的結果以及瀏覽器成功運行的畫面。

Docker Swarm & Stack

為什麼需要 Docker Swarm？

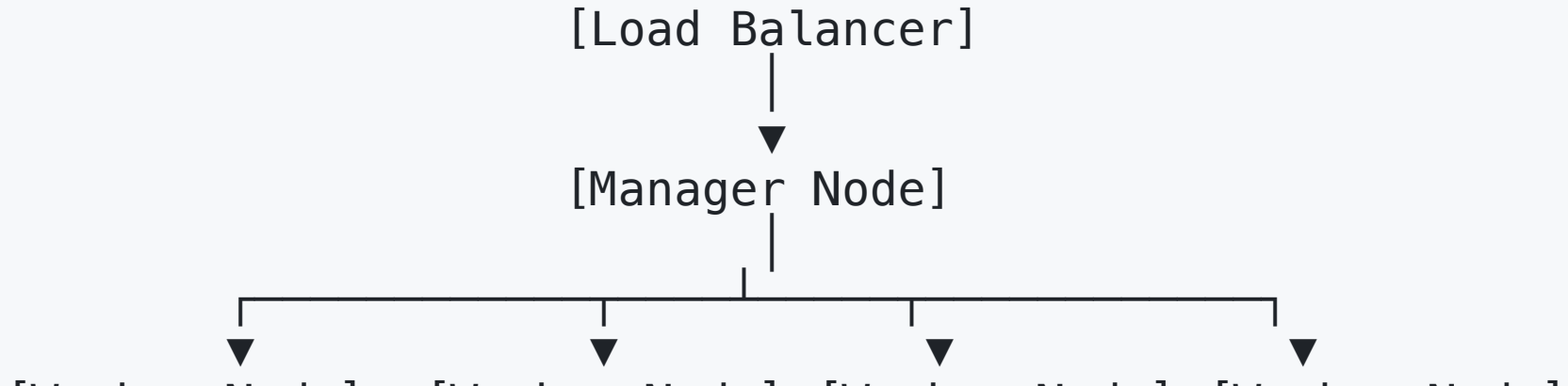
想像一下，你的應用程式需要更新：

- 如果直接停止舊版本再啟動新版本，會造成服務中斷
- 如果有多個使用者同時使用，這會造成不好的體驗
- 如果更新出問題，需要快速 Rollback

Docker Swarm 可以幫你實現：

- 零停機時間更新（Downtime-free updates）
- 自動負載平衡（Load balancing）
- 故障自動轉移（Failover）
- 服務自動擴展（Auto-scaling）

Docker Swarm 架構



當你要更新應用程式時：

1. Swarm 會先啟動新版本的容器
2. 確認新容器健康後，才關閉舊容器
3. 如果新版本有問題，可以立即回滾
4. 整個過程使用者無感

常用指令

```
# 初始化 Swarm
docker swarm init

# 部署 stack（零停機更新）
docker stack deploy -c docker-compose.yml todo-app

# 查看服務狀態
docker service ls

# 擴展服務數量（增加 Availability + Load Balancing）
docker service scale todo-app_frontend=3
```

> 雖然 Docker Swarm 可以處理基本的零停機部署，但在需要更複雜的部署策略時，可以考慮使用 Kubernetes。

雲端原生 (Cloud Native)

從本地到雲端：為什麼要部署到伺服器？

我們的應用程式現在可以在本機用 Docker Compose 順利執行了。但...

- 只有你自己能用。
- 如果關掉電腦，服務就停了。
- 如何讓其他人 (例如：使用者、團隊成員) 也能存取？

答案：將應用程式部署到一台公開的伺服器上！

簡介雲端原生 (Cloud Native)

雲端原生 (Cloud Native) 是一種建構和執行應用程式的方法，旨在充分利用雲端運算環境的優勢，實現快速、可靠、可擴展的應用程式交付。

核心概念與目標

- **容器化 (Containers)**：如 Docker，提供標準化的封裝與執行環境。
- **微服務 (Microservices)**：將應用拆分為小型、獨立、可獨立部署的服務。
- **持續整合/持續交付 (CI/CD)**：自動化建置、測試、部署流程。
- **DevOps 文化**：強調開發與維運團隊的協作、溝通與共同責任。
- **目標**：提升敏捷性 (Agility)、可擴展性 (Scalability)、彈性 (Resilience)、可觀測性 (Observability)。

更廣闊的雲原生生態 (未來展望)

- **容器編排 (Orchestration)**：如 Kubernetes，自動化容器的部署、擴展、管理。
- **服務網格 (Service Mesh)**：如 Istio, Linkerd，管理服務間通訊、安全、監控。
- **無伺服器 (Serverless)**：如 AWS Lambda, Azure Functions，專注於程式碼邏輯，無需管理底層伺服器。

> 今天，我們將體驗「雲端原生」中最基礎的一步：**將我們的容器化應用部署到一台遠端伺服器上**。

我們的「雲端」：工作站伺服器

今天，我們會使用一台預先設定好的 **工作站 Linux 伺服器 (VM)** 來模擬雲端主機。

目標

將我們的 Todo App (使用 Docker Compose) 部署到這台伺服器上，並能透過伺服器的 IP 位址公開存取。

> 把這台伺服器想像成你在 AWS EC2, Google Cloud VM, Azure VM 上租用的一台虛擬主機。

實作時間：幫大家開一個帳號連線到伺服器

任務

- 請依照教學文件指示，並跟隨助教的引導完成操作。
- 首先，助教會幫各組組長開一個帳號，並讓組長們連線到伺服器。
- 組長們連線到伺服器後，會再幫助組員們開帳號，並讓組員們連線到伺服器。

現在，每個人都可以連線到伺服器了！

連線到伺服器：SSH 複習

我們在 Week 12 已經學過如何使用 SSH 金鑰連線到 GitHub。同樣的原理也適用於連線到任何遠端伺服器。

連線指令

```
ssh your_username@server_ip_address
```

- `your_username`：登入伺服器的使用者名稱 (助教提供)。
- `server_ip_address`：伺服器的 IP 位址 (助教提供)。

> 第一次連線時，可能會詢問是否信任該主機的指紋，輸入 `yes` 即可。
> 如果設定了 SSH Key，則不需要輸入密碼。否則，會提示輸入密碼。

設定 SSH 金鑰

任務

- 請依照教學文件指示，並跟隨助教的引導完成操作。
- 設定 SSH 金鑰，並使用 SSH 連線到伺服器。

提示

- 自己電腦上運行 `ssh-keygen` 指令，生成 SSH 金鑰。
- 使用 `ssh-copy-id` 指令，將公鑰複製到助教的伺服器上。
- 同時伺服器會將公鑰加入 `~/.ssh/authorized_keys` 檔案中。
- 下次連線時，就不需要輸入密碼。

伺服器基本操作 (Linux)

成功登入伺服器後，你會進入 Linux 的命令列環境。以下是一些常用指令：

- `ls`: 列出目前目錄的檔案和資料夾。
  - `ls -a`: 顯示隱藏檔案。
  - `ls -l`: 顯示詳細資訊。
- `cd <directory_name>`: 切換目錄。
  - `cd ..`: 回到上一層目錄。
  - `cd ~` 或 `cd`: 回到家目錄。
- `pwd`: 顯示目前所在路徑。

- `mkdir <directory_name>`: 建立新資料夾。
- `rm <file_name>`: 刪除檔案。
  - `rm -r <directory_name>`: 刪除資料夾及其內容 (請小心使用！)。
- `cat <file_name>`: 查看檔案內容。
- `nano <file_name>` 或 `vim <file_name>`: 文字編輯器 (nano 較易上手)。
- `sudo <command>`: 以系統管理員權限執行指令 (請謹慎使用)。
- `df -h`: 查看磁碟空間使用情況。
- `free -m`: 查看記憶體使用情況。
- `top` 或 `htop`: 查看系統程序與資源使用情況。

實作時間：部署 Todo App 到工作站伺服器

任務

- SSH 登入助教提供的工作站伺服器。
- Clone 你的 Todo App 專案。
- 使用 `docker-compose` 在伺服器上啟動你的應用。
- 從你本地的瀏覽器，透過伺服器 IP 成功存取你的 Todo App。

跟著做

- 請依照教學文件指示，並跟隨助教的引導完成操作。
- **點名**：瀏覽器成功顯示從伺服器運行的 Todo App 畫面，以及伺服器上 `docker-compose ps` 的輸出。

重要「真實世界」考量 (簡述)

我們今天的部署非常基礎，真實世界的雲端部署還需要考慮：

- **安全性**：
  - 防火牆設定 (只開放必要的 port，例如 `ufw` on Linux)。
  - HTTPS 加密 (SSL/TLS 憑證，例如 Let's Encrypt)。
  - 環境變數與密鑰管理 (例如 Vault, Doppler, .env 檔案配合 .gitignore)。
  - 定期更新系統與軟體套件。
  - 使用非 root 使用者執行應用程式。
- **網域名稱 (Domain Name)**：用好記的網址 (如 `www.mytodoapp.com`) 而非 IP 位址，並設定 DNS 紀錄。
- **資料庫持久化與備份**：確保資料庫資料在容器重啟後不會遺失，並定期備份。

- **監控與告警 (Monitoring & Alerting)**：追蹤應用程式效能 (APM)、錯誤、伺服器資源，並在出問題時通知 (例如 Prometheus, Grafana, Sentry)。
- **日誌管理 (Logging)**：集中管理與分析應用程式及系統日誌 (例如 ELK Stack, Loki)。
- **基礎設施即程式碼 (Infrastructure as Code, IaC)**：使用程式碼管理和配置基礎設施 (例如 Terraform, Ansible)。
- **高可用性與擴展性 (High Availability & Scalability)**：設計系統以應對故障並能根據負載自動擴展。

> 這些是更進階的主題，未來有機會可以深入學習。

總結與下一步

本週回顧 (Week 13)

- **Docker Compose**：簡化多容器應用管理。
  - `docker-compose.yml` 的撰寫。
  - `docker-compose up`, `down`, `logs`, `ps` 等指令。
- **初探雲端部署**：
  - SSH 連線到遠端伺服器。
  - 在伺服器上使用 Git 和 Docker Compose 部署應用。
  - 透過公網 IP 存取應用。

下一週 (Week 14) 預告

- **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大概念。
- **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。

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！