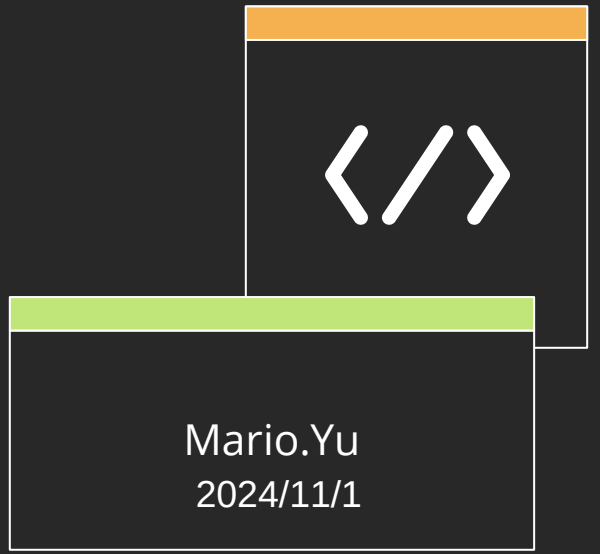
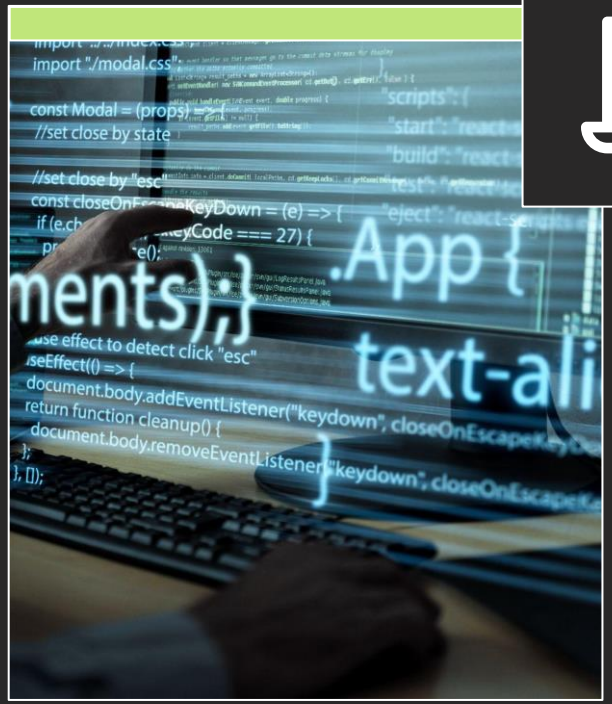
Dependency Injection



Mario.Yu

2024/11/1

目錄


import "./modal.css";
const Modal = (props) => {
//set close by state
//set close by "esc"
const closeOnEscapeKeyDown = (e) => {
if (e.keyCode === 27) {
ments),}
.App {
text-ali
use effect to detect click "esc"
useEffect(() => {
document.body.addEventListener("keydown", closeOnEscape
return function cleanup() {
document.body.removeEventListener("keydown", closeOnEscape
}, []);
"scripts": {
"start": "react-
"build": "react-
"test": "react-
"eject": "react-scripts eject"

01

依賴基本概念

需求

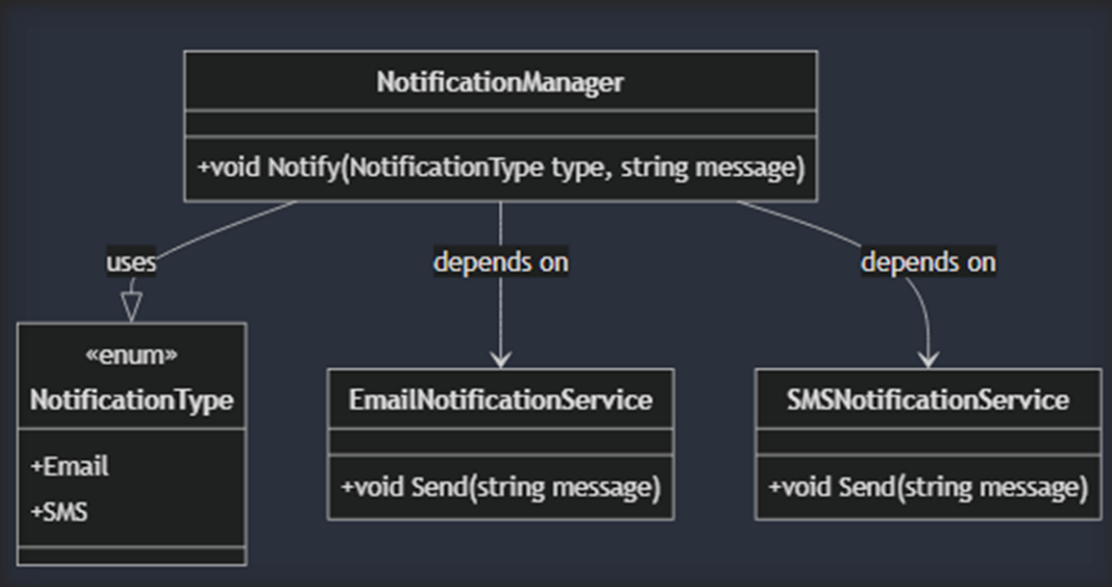
公司有一個簡單的通知系統，主要用於向客戶發送通知消息。最初系統僅支持 Email 通知，所以 NotificationManager 類別直接依賴 EmailNotificationService 的實作。然而，隨著業務擴展，客戶希望能夠選擇 SMS 通知，甚至未來還有可能增加 Push Notification、Line 訊息通知等多種渠道。

Pair Programming

請實作SMSNotificationService

依賴概念

沒有使用介面的設計



需求

公司有一個簡單的通知系統，主要用於向客戶發送通知消息。最初系統僅支持 Email 通知，所以 NotificationManager 類別直接依賴 EmailNotificationService 的實作。然而，隨著業務擴展，客戶希望能夠選擇 SMS 通知，甚至未來還有可能增加 Push Notification、Line 訊息通知等多種渠道。

Pair Programming

請實作SMSNotificationService
並使用介面實作

依賴概念

使用介面的設計

蕭昱翔表演時間

依賴概念

沒有使用介面的設計

依賴反轉(DIP)

- 高階模組與低階模組之間的直接依賴解耦，並讓高階模組與低階模組都依賴於一個抽象介面。

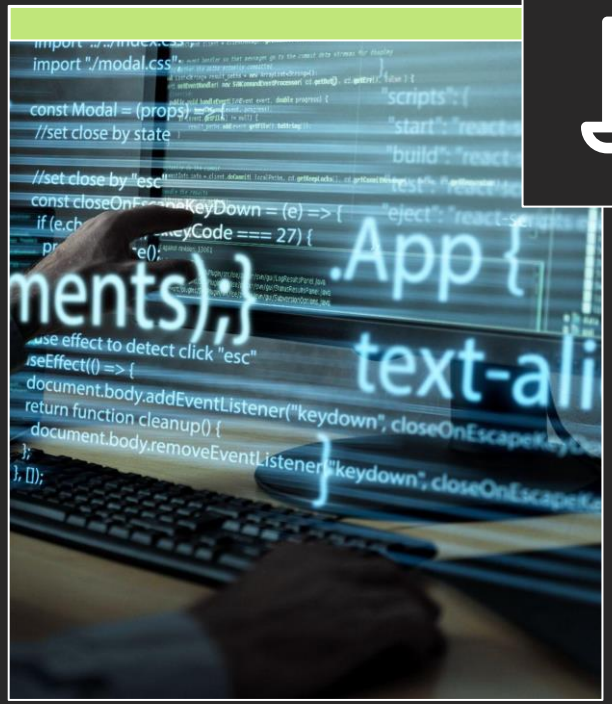
- ✓ **增加系統的可測試性**，依賴介面或抽象時，可以輕鬆替換底層實作來進行單元測試。
- ✓ **提升模組間的鬆耦合**，設計可以使模組間的依賴關係減少，讓系統更具彈性抽換。
- ✓ **支援依賴注入**，簡化組件管理。

高階模組通常關注於整體行為，它們定義「做什麼」，而不是「怎麼做」。而低階則是具體細節操作。

依賴 vs 耦合

依賴指的是**一個類別需要另一個類別的功能**或資源來完成自己的工作。也就是說，A 類別「依賴於」 B 類別，因為 A 無法獨立運行，必須依賴 B 所提供的功能。依賴通常會透過參數、屬性或方法呼叫等方式來表現。

耦合則是**相依後產生的影響性，表相依關係的依賴程度**。簡單來說A物件參考B物件後，當B物件有異動會一定程度的影響，此時就需要做一些解偶手段，例如常見的解偶技巧就是使用介面隔離耦合或是服務層級的解耦Mediator。



1 0 1 0 1 0 1 1 1 0 0 0 1 1 1 0

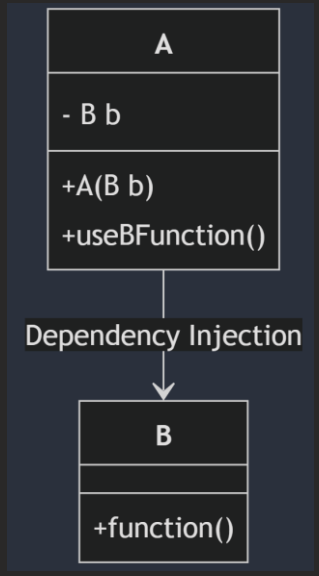
1 1 0 0 1 0 1 0 1 0 1 0 1 1 1 0

02

DI的注入模式

注入

Injection 通常指的就是將一個 **物件實例（Instance）** 傳入**另一個物件**中，以解耦兩者的依賴關係



? 無法靈活變更：如果未來需要替換 B，就必須修改 A。

? 測試困難：當 *A* 直接依賴 *B*，進行單元測試時就必須依賴真實。

? 高耦合度： *A* 和 *B* 的緊密相依會導致程式複雜度增加，且不利於擴充和維護。

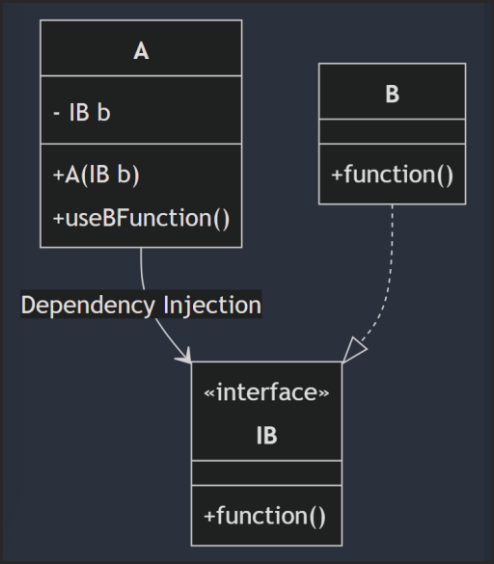
✓ 彈性提升：可以隨時替換 B 的實現，而不影響 A。

✓ 測試容易：可以在測試中替換為模擬（mock）物件，讓測試更輕鬆。

✓ 擴展性高：系統架構變得更具彈性和易於維護。

注入

Injection 通常指的就是將一個 物件實例（Instance） 傳入另一個物件中，以解耦兩者的依賴關係



? 無法靈活變更：如果未來需要替換 B，就必須修改 A。

? 測試困難：當 *A* 直接依賴 *B*，進行單元測試時就必須依賴真實。

? 高耦合度： *A* 和 *B* 的緊密相依會導致程式複雜度增加，且不利於擴充和維護。

✓ 彈性提升：可以隨時替換 B 的實現，而不影響 A。

✓ 測試容易：可以在測試中替換為模擬（mock）物件，讓測試更輕鬆。

✓ 擴展性高：系統架構變得更具彈性和易於維護。

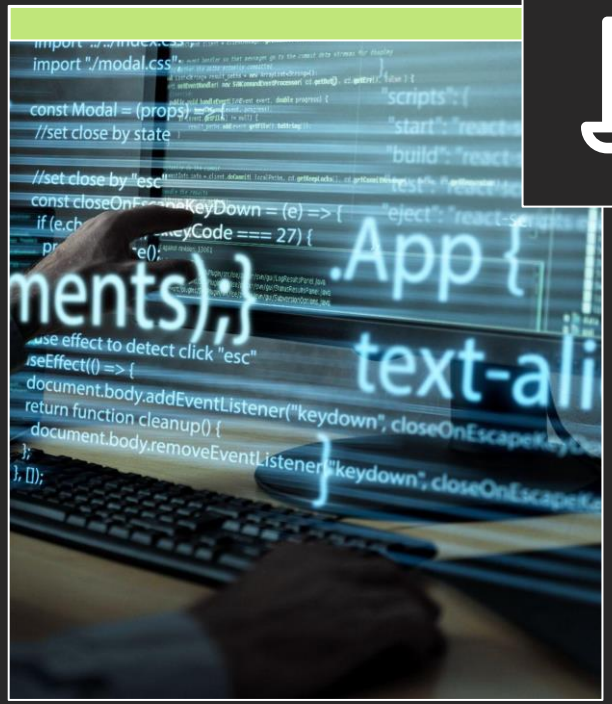
注入三種方式

1. 建構子注入（Constructor Injection）
2. 屬性注入（Property Injection）
3. 方法注入（Method Injection）

方式	優點	缺點
建構子注入	• 確保依賴注入 • 依賴明確	• 依賴過多時建構子複雜 • 不適合可選依賴
屬性注入	• 靈活性高 • 適合可選依賴	• 可能發生空指標異常 • 依賴可被隨意修改
方法注入	• 精確控制依賴 • 適合臨時依賴	• 難以管理依賴狀態 • 測試較複雜

窮人的注入

使用者手動創建以new物件的方式，各自注入，除了麻煩外而且會有傳遞注入等等的問題。


import "./modal.css";
const Modal = (props) => {
//set close by state
//set close by "esc"
const closeOnEscapeKeyDown = (e) => {
if (e.keyCode === 27) {
use effect to detect click "esc"
useEffect(() => {
document.body.addEventListener("keydown", closeOnEscape);
return function cleanup() {
document.body.removeEventListener("keydown", closeOnEscape);
}, []);
.App {
text-ali

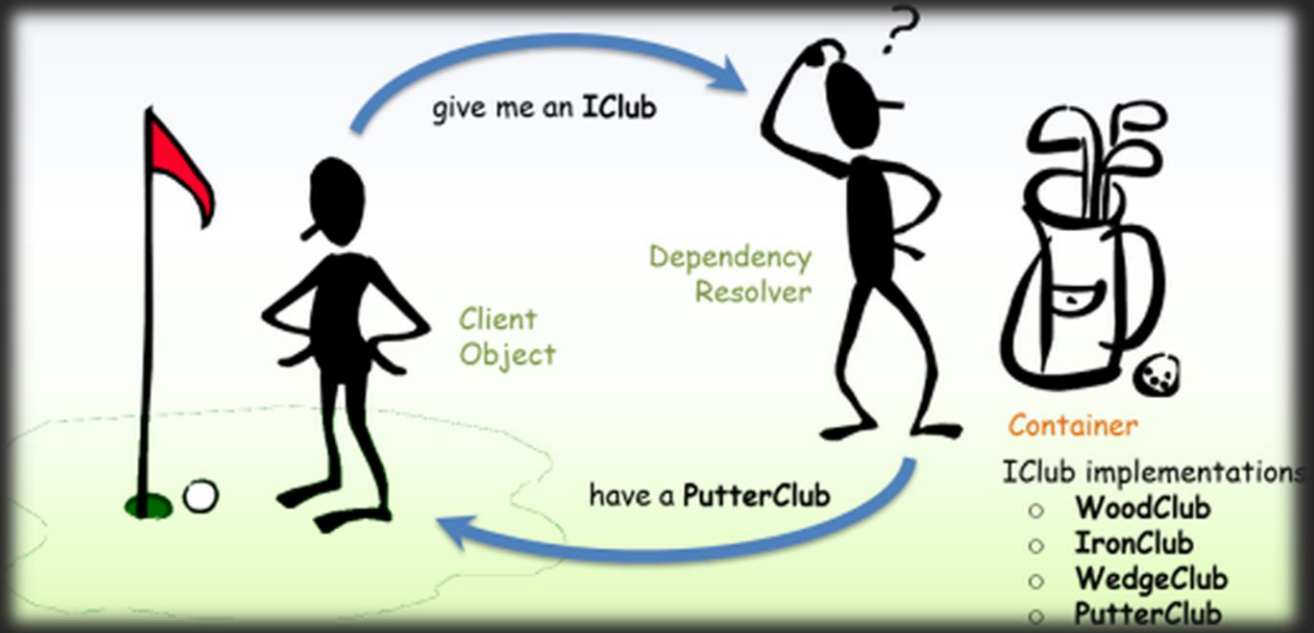
03

容器的概念與用途

DI Container

窮人打高爾夫，只能自己背球袋

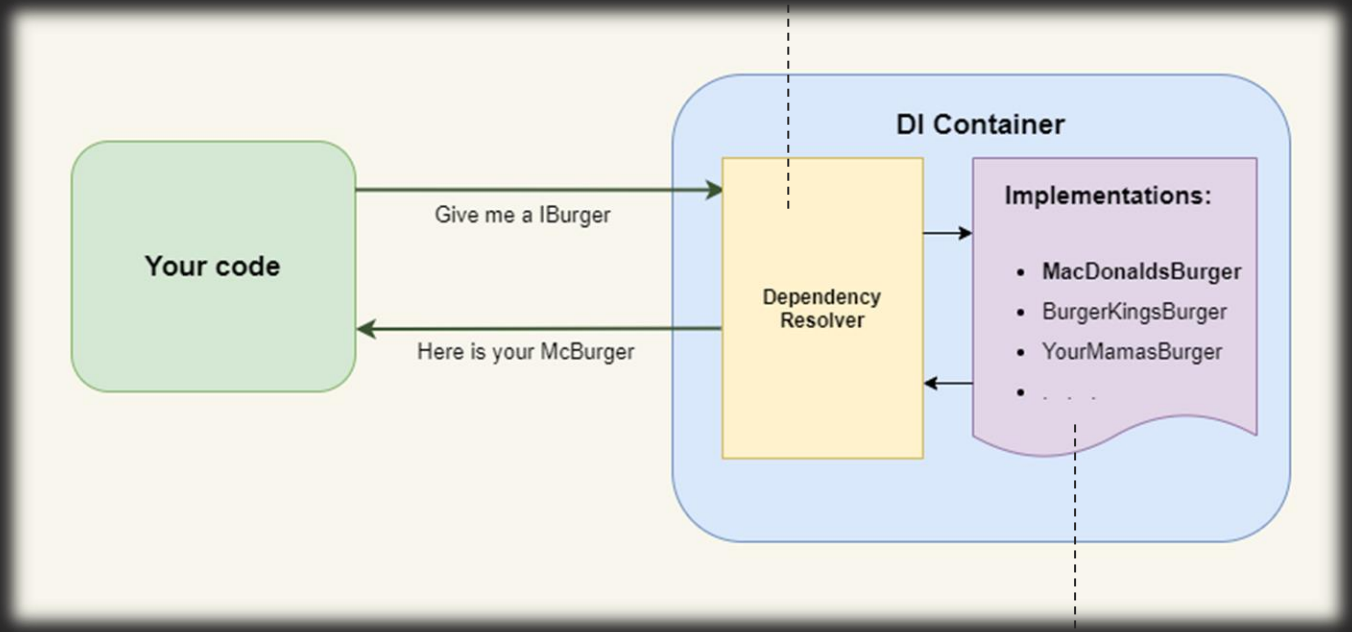
DI Container



DI框架就像一個球僮幫你背著球袋，事實給你對應球桿(物件)

DI Container

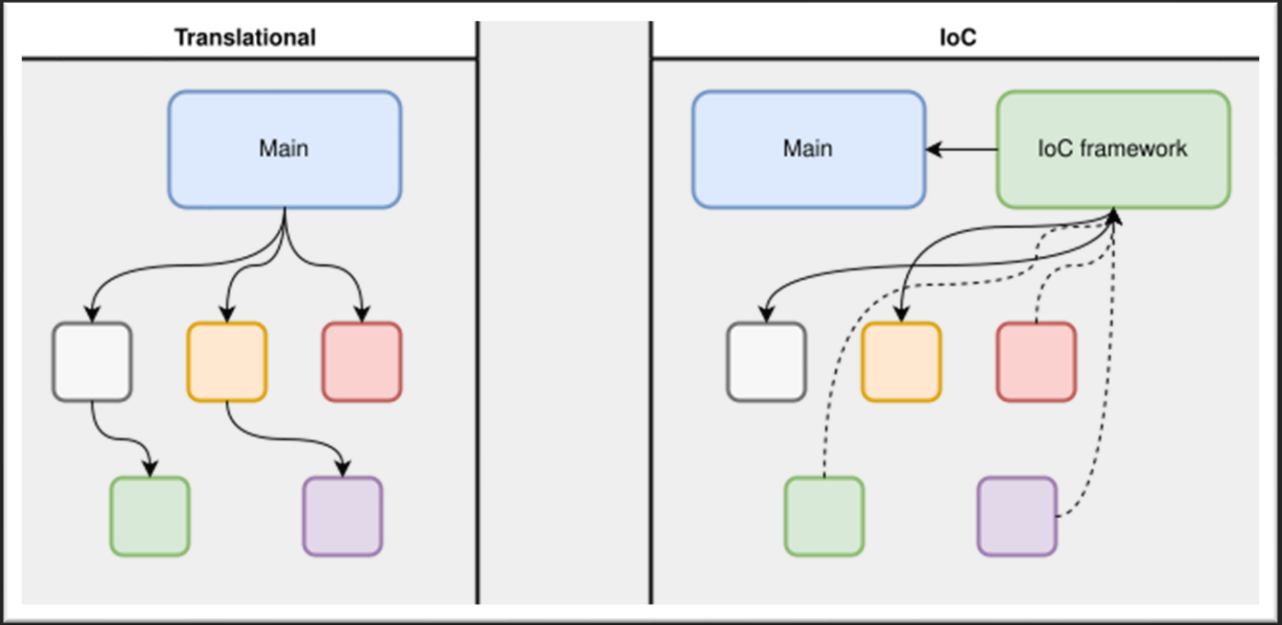
根據請求找到對應的具體實現類別

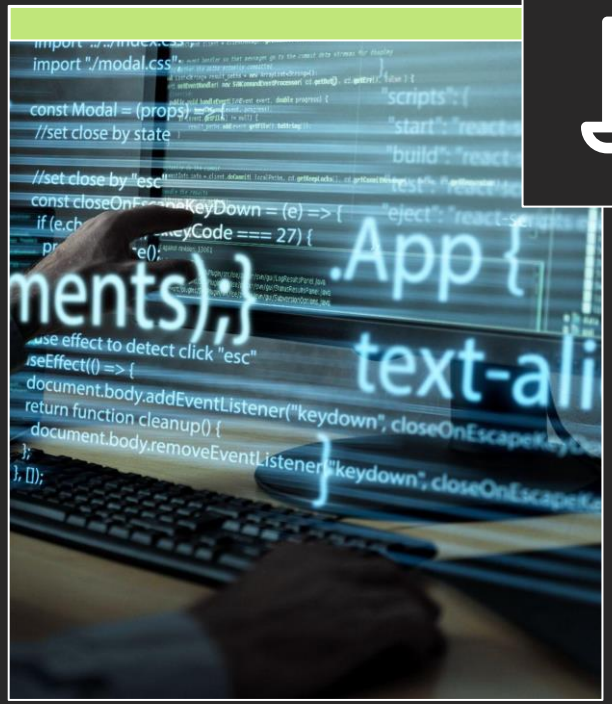


Mapping 表 或 註冊表

IOC(控制反轉)

你可以將IOC是一種設計模式，將流程控制重定向到外部處理程序或控制器來提供控制結果，而不是透過控制項(大多情境就是程序)來直接得到結果。





04

DI 物件生命時間

1 1 0 0 1 0 1 0 1 0 1 0 1 0 1 1 0

1 0 1 0 1 0 1 1 1 0 0 0 1 1 1 0

物件生命週期

```
public class Logger
{
    private static Logger _instance;
    private static readonly object _lock = new object();
```

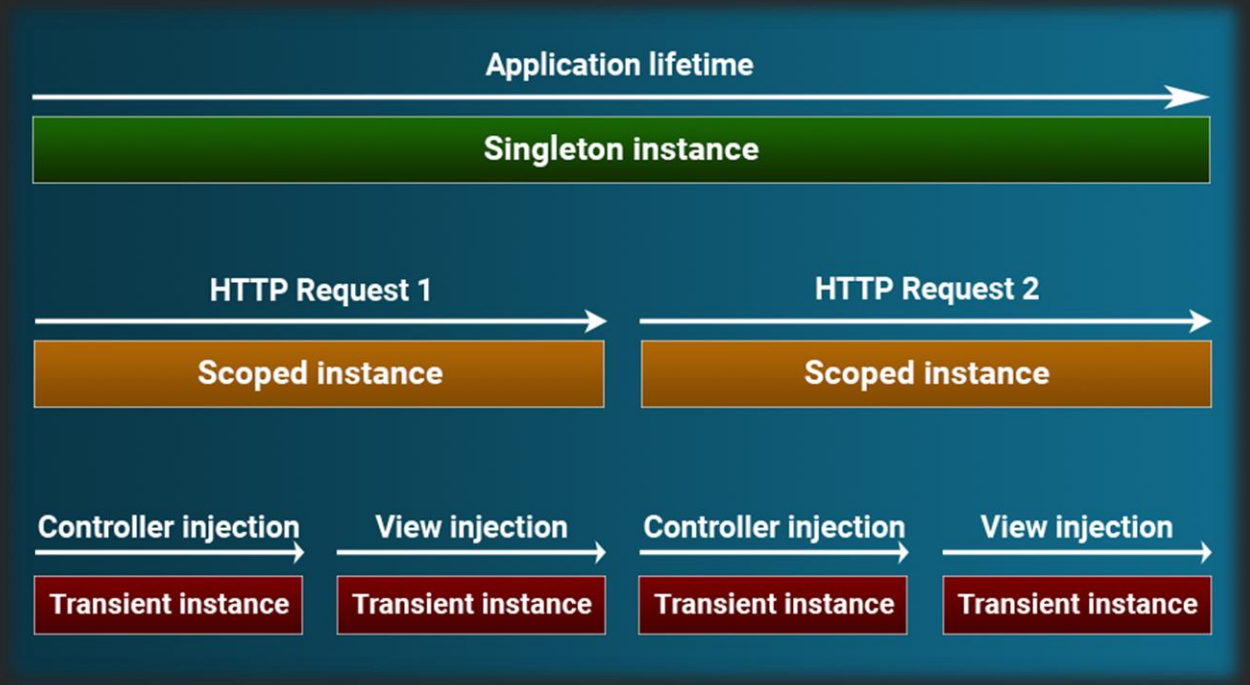
物件生命週期

```
public class SomeService
{
    public void DoSomething()
    {
        var repository = new Repository(); // 每次呼叫都會新建 Repository
        repository.GetData();
    }
}
```

物件生命週期

```
public void ProcessData()
{
    using (var repo1 = new Repository())
    {
        repo1.BeginTransaction();

        var repo2 = new Repository(); // repo2 無法與 repo1 共用同一交易
        repo1.UpdateData();
        repo2.InsertData();

        repo1.Commit(); // repo2 的資料不會受到 repo1 的交易控制
    }
}
```

DI 物件生命時間

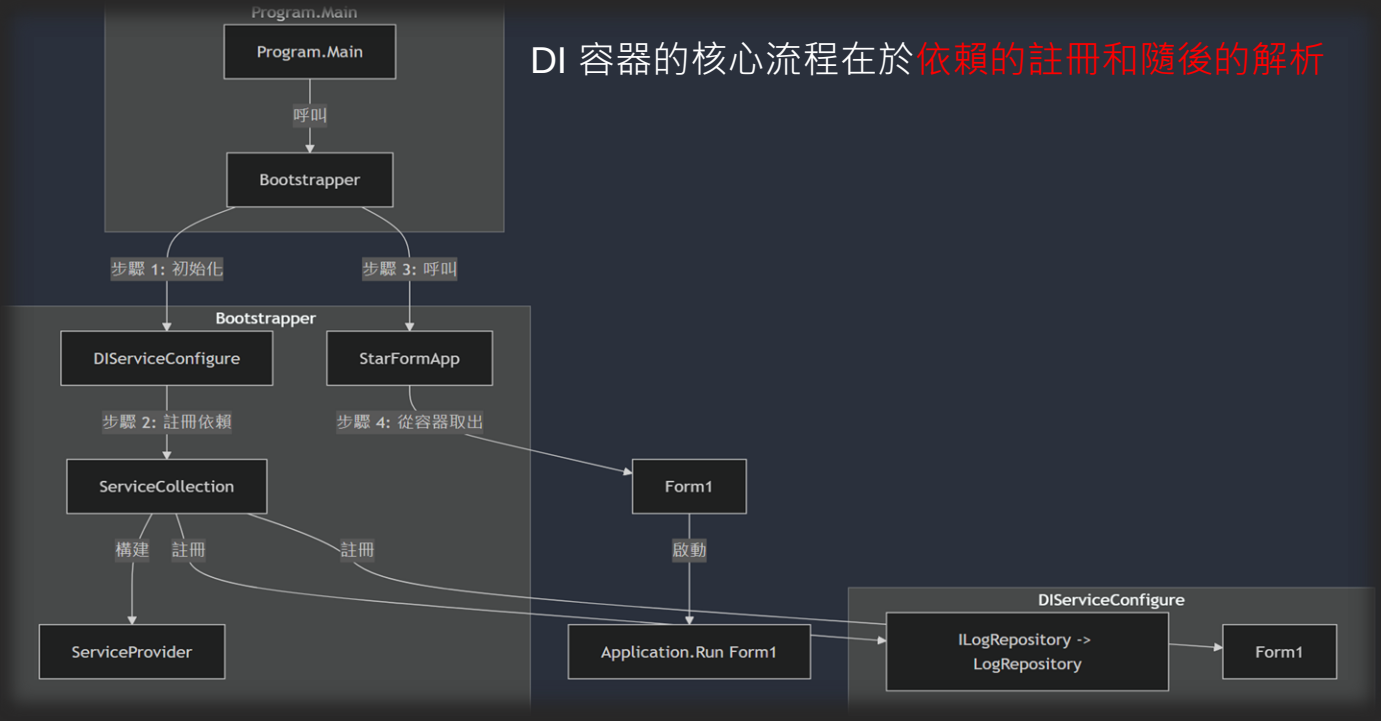

Application lifetime
Singleton instance
HTTP Request 1
Scoped instance
HTTP Request 2
Scoped instance
Controller injection
Transient instance
View injection
Transient instance
Controller injection
Transient instance
View injection
Transient instance

DI容器實際操作

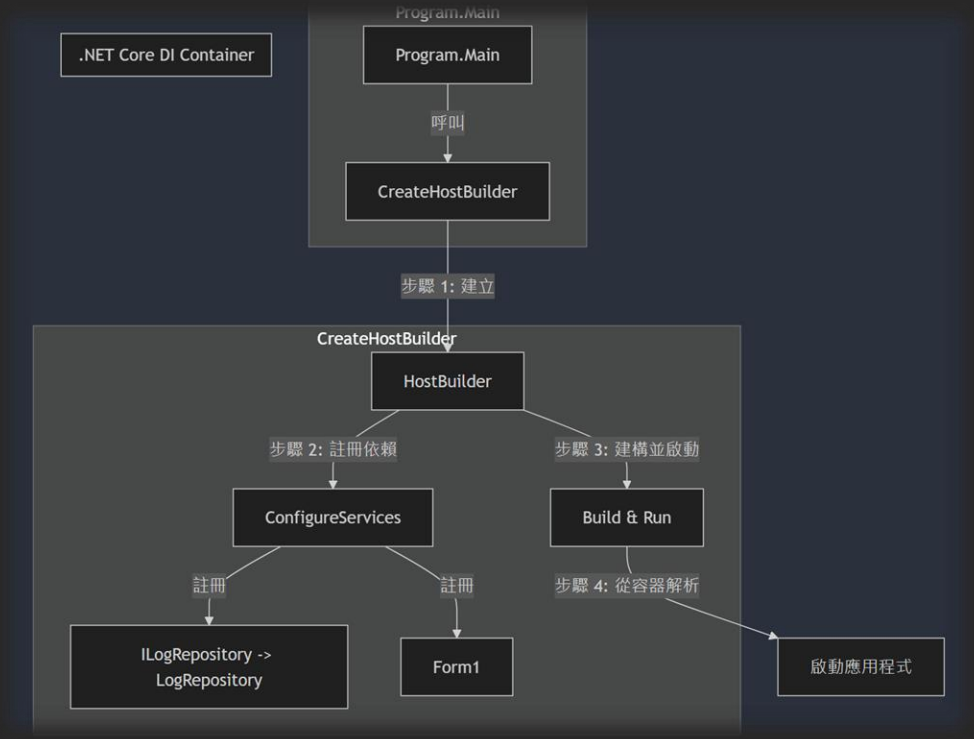
```
// 建立服務容器
var serviceCollection = new ServiceCollection();

// 設定生命週期（改成 AddSingleton 或 AddScoped 來比較）
serviceCollection.AddScoped<CounterService>();

//  Factory Pattern（工廠模式） 的一種應用。
//  記憶體中建立一個「註冊規則」的記錄
var serviceProvider = serviceCollection.BuildServiceProvider();
```

How to use

DI 容器的核心流程在於依賴的註冊和隨後的解析



How to use


.NET Core DI Container
Program.Main
Program.Main
呼叫
CreateHostBuilder
步驟 1: 建立
CreateHostBuilder
HostBuilder
步驟 2: 註冊依賴
步驟 3: 建構並啟動
ConfigureServices
Build & Run
註冊
註冊
步驟 4: 從容器解析
ILogRepository ->
LogRepository
Form1
啟動應用程式

Thanks!

Do you have any questions?
pal2097@gmail.com