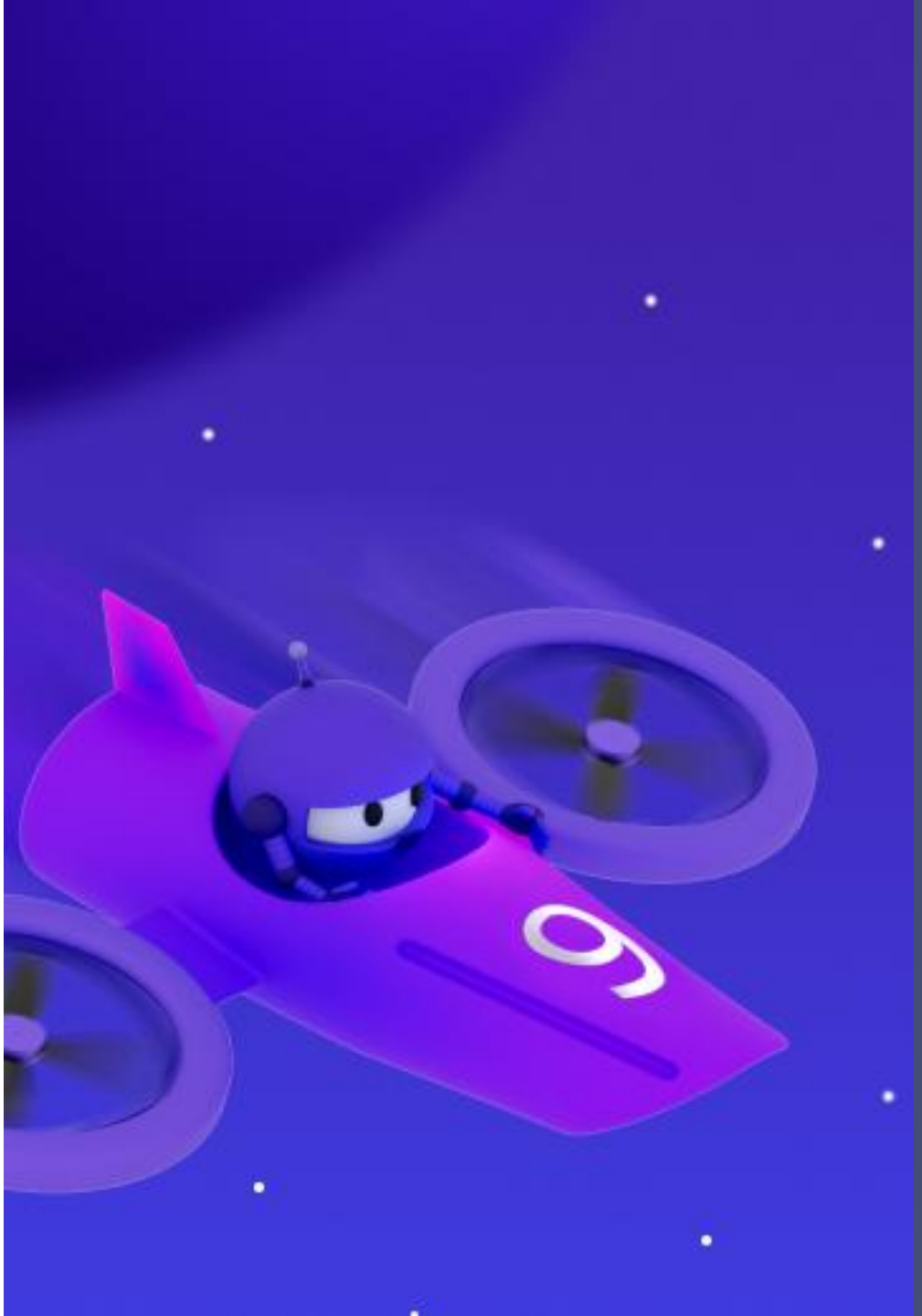
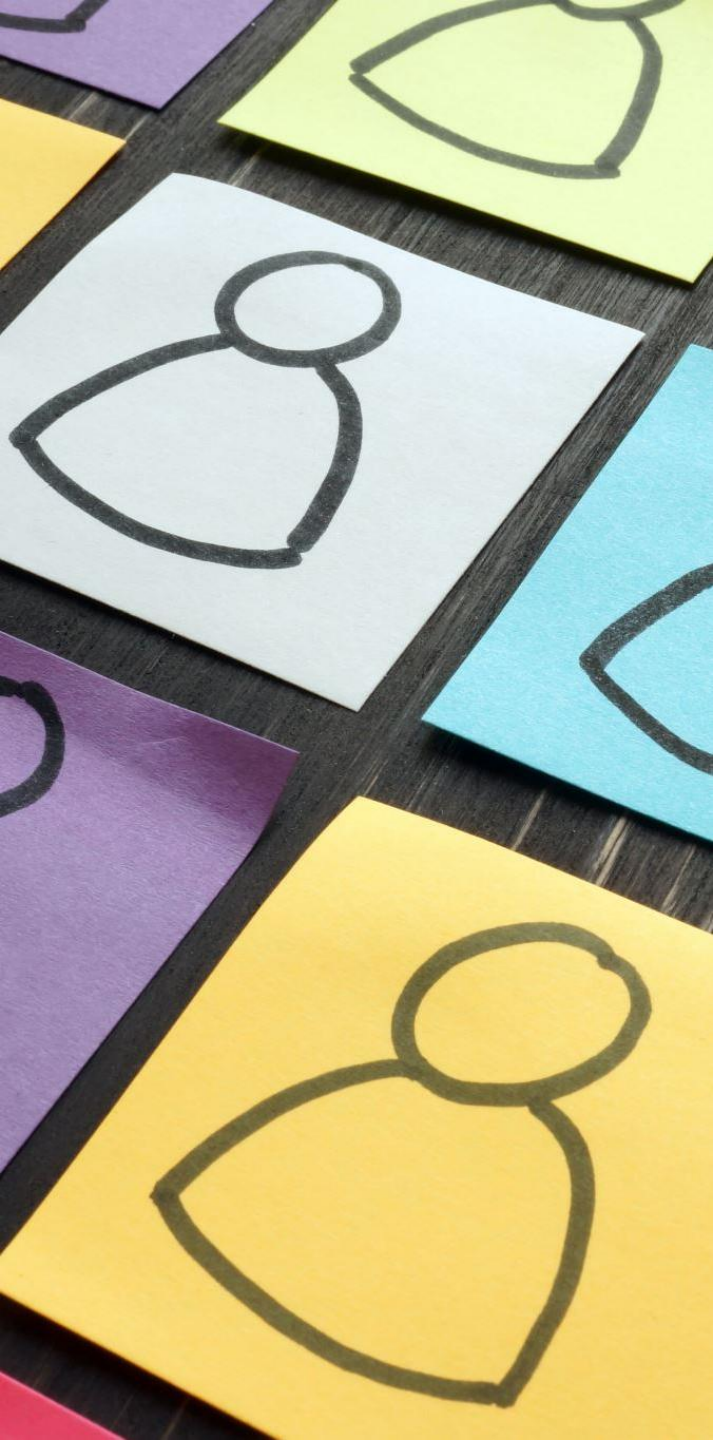


# Hello, World를 Generic Host로 다시 쓰기

남정현

닷넷데브, Microsoft MVP



# 오늘 살펴볼 내용

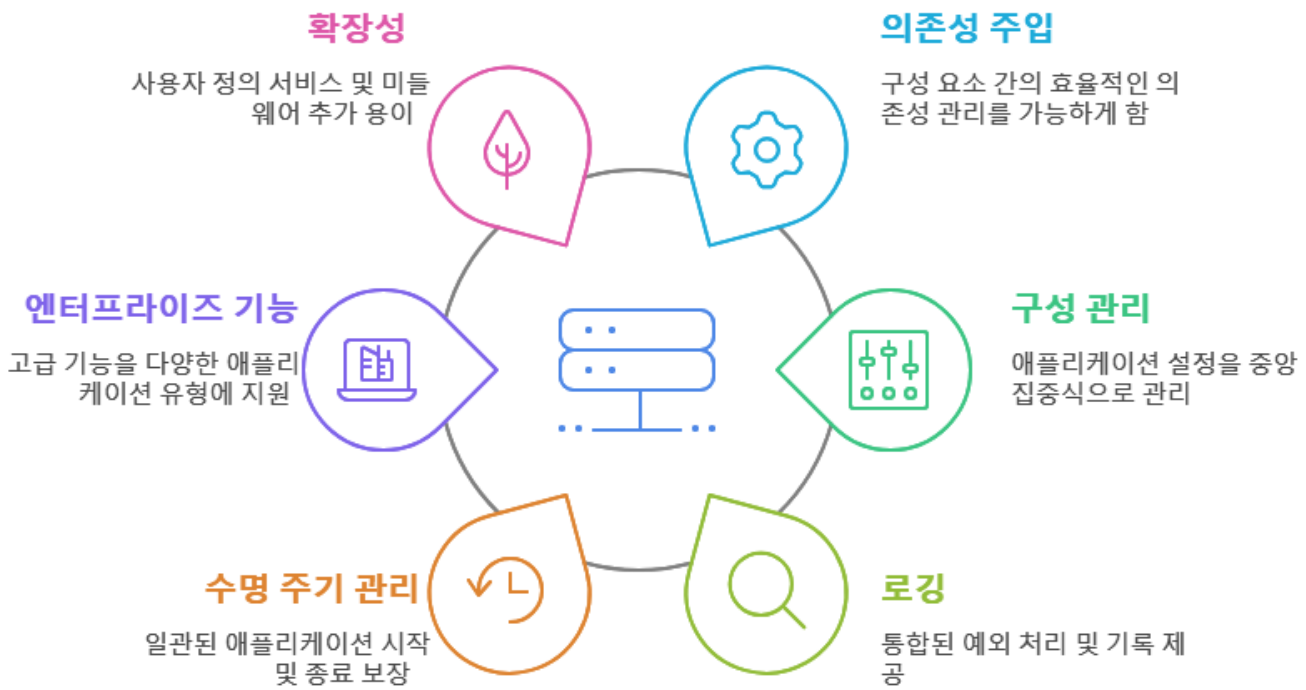
- Microsoft.Extensions.Hosting 소개
- 닷넷 애플리케이션 호스팅 모델
- 우리가 아는 Hello, World!의 문제점
- 제네릭 호스트로 Hello, World! 다시 쓰기
- 종속성 주입과 구성 활용하기
- 마무리

# MODERN .NET에 대하여

- .NET 5 이후의 최신 닷넷 기술을 여러분은 얼마나 알고 계신가요?
- Modern .NET을 정의하는 핵심적인 요소는 바로 App Host 모델입니다.
- App Host 모델은 범용 Generic 모델부터, ASP .NET Core, MAUI, Blazor, Aspire 등 다양한 곳의 기반 기술로 쓰입니다.
- App Host 모델의 초기화 코드는 의존성 주입을 기반으로 App Builder 패턴을 사용하여 애플리케이션을 구성합니다.
- 20년전에 쓰인 Hello, World! 샘플 코드를 Modern .NET 스타일로 고쳐 쓰고, 프로덕션에서 활용하는 방법을 오늘 소개합니다.

# Microsoft.Extensions.Hosting 소개

# 앱 호스트를 사용하는 여섯 가지 이유



# SPRING과 MODERN .NET 사이의 비교

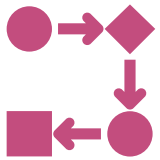
- .NET App Host는 독자적인 모델을 갖추고 있습니다.
- Spring Application, Spring Container와 대응되는 기능을 .NET에서 찾으신다면, Microsoft.Extensions.Hosting이 그 답입니다.
- 컨테이너, 의존성 주입, 생명 주기 관리, 구성 관리 등 여러 세부 요소들을 하나씩 비교해보겠습니다.

# SPRING VS MODERN .NET

| | Spring | Modern .NET |
|---|---|---|
| 컨테이너 | Application Context가 Bean 컨테이너로 작동 | Service Collection이 Service 컨테이너로 작동 |
| 의존성 주입 | @AutoWired, @Inject 등의 어노테이션 사용 | 생성자 주입, IServiceProvider, ActivationUtilities 사용 |
| 생명 주기 관리 | Bean Scope (Singleton, Prototype 등) | Service Lifetime (Singleton, Transient, Scoped) |
| 초기 부트스트랩 | Spring Boot CLI | Dotnet CLI |
| 패키지 생태계 | Maven Central | NuGet.org |
| 빌드 시스템 | Maven, Gradle, Ivy | Dotnet CLI, MSBUILD |
| 관점 지향 프로그래밍 (AOP) | @Aspect 어노테이션 | 미들웨어, 정적 소스 생성기 |
| 웹 애플리케이션 | MVC, WebFlux | ASP .NET Core (MVC, Minimal API), Blazor Sever + async/await, Rx.NET |

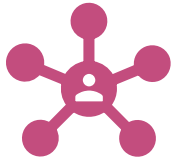
# 닷넷 애플리케이션 호스팅 모델

# 호스팅 모델의 정의



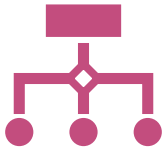
### 애플리케이션의 수명 주기 관리

시작부터 종료까지 전체 생명주기를
체계적으로 제어하는 프레임워크 구조



### 리소스 관리 담당

메모리, 네트워크 연결, 파일 시스템 등의
리소스를 효율적으로 할당하고 해제



### 체계적인 실행 흐름 제어

비동기 작업, 취소 토큰, 예외 처리 등을
일관된 방식으로 관리

# 전통적인 콘솔 애플리케이션과의 차이

- 구조화된 애플리케이션 시작/실행/종료
  - 명확한 진입점과 종료 지점, 우아한 종료 처리 지원
- 크로스커팅 관심사의 통합 관리
  - 로깅, 설정, 의존성 주입 등을 중앙 집중식으로 처리
- 확장 가능한 아키텍처 제공
  - 모듈식 구조로 새로운 기능을 쉽게 추가하고 확장 가능

# MICROSOFT.EXTENSIONS.HOST의 역할

- 호스팅 모델의 구현체 제공
  - 즉시 사용 가능한 표준 호스팅 인프라 제공
- 엔터프라이즈급 기능 지원
  - 마이크로서비스, 워커 서비스 등 현대적인 애플리케이션 패턴 구현
- 일관된 애플리케이션 구조 제공
  - ASP.NET Core와 동일한 패턴을 콘솔 애플리케이션에서도 사용

# 우리가 아는 Hello, World!의 문제점

# 우리가 잘 아는 코드입니다. 그렇죠?

# 그런데 무슨 문제가 있을까요?

```
internal static class Program
{
  private static void Main(string[] args)
  {
    // Q: 만약 환경 변수가 아닌 JSON 파일로 설정을 하고 싶다면 어떻게 고쳐야 할까요?
    var language = Environment.GetEnvironmentVariable("LANGUAGE");

    // Q: 만약 특정한 언어가 지원되는지 테스트하는 테스트 코드를 만들고 싶다면 어떻게 고쳐야 할까요?
    var message = ((language ?? string.Empty).Trim().ToUpperInvariant()) switch
    {
      "KO" => "안녕, 세계!",
      "JA" => "こんにちは世界！",
      _ => "Hello, World!",
    };

    // Q: 만약 콘솔 화면이 아닌 파일로 메시지를 출력하거나, 다른 웹 API로 메시지를 보내고 싶다면 어떻게 고쳐야 할까요?
    Console.Out.WriteLine(message);
  }
}
```

# 여러분이라면 어떻게 하시겠습니까?

- Q: 만약 환경 변수가 아닌 JSON 파일로 설정을 하고 싶다면 어떻게 고쳐야 할까요?
- Q: 만약 특정한 언어가 지원되는지 테스트하는 테스트 코드를 만들고 싶다면 어떻게 고쳐야 할까요?
- Q: 만약 콘솔 화면이 아닌 파일로 메시지를 출력하거나, 다른 웹 API로 메시지를 보내고 싶다면 어떻게 고쳐야 할까요?
- 여기에 대한 답이 Microsoft.Extensions.Hosting에 있습니다.

# 단순한 구조로 인한 제약사항

의존성 주입이나 구성 관리의 어려움

체계적인 애플리케이션 수명 주기 관리의 부재

크로스커팅 관심사(로깅, 오류 처리 등) 처리의 한계

# 확장성과 유지보수의 어려움

기능 추가 시 코드 구조 변경이 필요

일관된 패턴 적용이 어려움

테스트 용이성 부족

# 엔터프라이즈 환경에서의 한계

설정 관리의 유연성 부족

복잡한 요구 사항을 맞추기 위한 기능 구현의 복잡성

모니터링과 진단 기능 통합의 어려움

# 제네릭 호스트로
# Hello, World! 다시 쓰기

# 기본 프로젝트 구조

Microsoft.Extensions.Hosting NuGet 패키지 참조 추가

Program.cs에서 AppHost 구성

WorkerService를 만들고, App Lifetime 제어

# 새 프로젝트 만들기

- dotnet new console -n NewHelloWorld
- cd ./NewHelloWorld
- dotnet add package Microsoft.Extensions.Hosting
- code .

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

var builder = Host.CreateApplicationBuilder(args);
builder.Services.AddHostedService<MainApp>();

var app = builder.Build();
app.Run();
```

# 기존 MAIN 메서드 코드 변경

- App Host 생성 후, 컨테이너에 백그라운드 서비스 + 각종 서비스 추가 (Wire Config)
- 종료 시그널이 발생하기까지 계속 실행되도록 Run 메서드로 애플리케이션 실행

```
public class MainApp(
    IHostApplicationLifetime Lifetime
) : BackgroundService
{
    protected override Task ExecuteAsync(CancellationToken
    {
        try
        {
            Console.WriteLine("Hello, World!");
        }
        catch (Exception ex)
        {
            Console.Error.WriteLine(ex.ToString());
        }
        finally
        {
            Lifetime.StopApplication();
        }

        return Task.CompletedTask;
    }
}
```

# 시작 코드 만들기

- BackgroundService로 MainApp 로직 구현
- async/await 문이 없으면 Task.CompletedTask 반환
- async/await이 있으면 기존처럼 사용 가능
- 이 백그라운드 서비스가 종료되면 프로그램 전체 종료를 위해 Lifetime.StopApplication 호출

# 의존성 주입 설정

서비스 수명주기 (Singleton, Scoped, Transient) 관리

인터페이스 기반 구현으로 느슨한 결합 구현

생성자 주입을 통한 의존성 해결

```
public interface IHelloWorld
{
    string GetMessage(string? language);
}

public class HelloWorld(
    IConfiguration Config
) : IHelloWorld
{
    public string GetMessage(string? language)
    {
      return (language ?? string.Empty).Trim().ToUpperInvariant() switch
      {
        "KO" => "안녕, 세계!",
        "JA" => "こんにちは世界！",
        _ => "Hello, World!",
      };
    }

    public string GetMessage()
        => GetMessage(Config["LANGUAGE"]);
}
```

# 추가할 서비스 - HELLO WORLD

- 정적 메서드 대신, HelloWorld를 클래스로 만들었습니다.
- 단순 문자열만 반환하는 HelloWorld와 나중에 다른 형태의 HelloWorld 버전을 구분하기 위해 인터페이스를 두었습니다.
- 생성자 주입을 이용해서 IConfiguration을 받습니다.

# 생명 주기 이해

Transient: GetService, GetRequiredService를 호출할 때 마다 객체를 만들려고 하는 경우

Singleton: 처음 GetService, GetRequiredService가 불렸을 때에만 만들고 이후에는 계속 재사용 하는 경우 – **이 예제의 HelloWorld는 Singleton이 적절합니다.**

Scoped: IServiceProvider.CreateScope 메서드로 만든 IServiceScope의 Dispose 메서드 (using, IDisposable)가 불리기 전 까지만 재사용하는 경우 (주로 ASP .NET에서 많이 사용됨)

# 종속성 주입, 구성 활용하기

# 로깅 통합

- ILogger 인터페이스를 통한 구조화된 로깅
- 로그 레벨 (Debug, Info, Warning, Error) 구분
- 다양한 로그 프로바이더 (Console, Debug) 지원

# 다양한 로깅 공급자

- builder.Logging.AddJsonConsole();
  - JSON 형식으로 모든 로그 기록을 출력 – ELK 스택을 위한 JSON 로그 수집에 바로 투입하기 위한 목적으로 활용 가능
- builder.Logging.AddSystemdConsole();
  - 리눅스 시스템 서비스로 프로세스를 실행할 때 시스템 로그로 남기기 위한 목적으로 콘솔 추가
- builder.Logging.AddEventLog();
  - Windows 이벤트 로그에 기록을 남기기 위한 목적으로 추가
- builder.Logging.AddConsole();
  - 보편적인 콘솔 로그 – 기본적으로 추가됨
- builder.Logging.AddDebug();
  - Visual Studio나 다른 라이브 디버거의 자체 콘솔에 출력할 수 있도록 기능 추가

# 구성 통합

기본 내장으로 환경 변수, JSON 파일 처리, 명령줄 처리 지원

개발자 컴퓨터에만 보관하는 시크릿 설정 분리, 인 메모리 설정 고정 지원

확장 패키지를 통해 HashiCorp Nomad, 혹은 기타 구성 관리 DB (CMDB)도 지원 가능

```
var builder = Host.CreateApplicationBuilder(args);

builder.Configuration.AddInMemoryCollection(
    new Dictionary<string, string?>
    {
        { "LANGUAGE", "KO" }
    }
);
```

# 내 마음대로 구성 바꾸기

- CreateApplicationBuilder 안에는 기본적인 로깅, 구성 서비스가 자동으로 포함되도록 설계되어 있음
- 만약 유닛 테스트가 필요한 경우, 빌더 수준의 구성에서 AddInMemoryCollection을 이용하여 원하는 설정 값을 코드 상에서 주입 가능

# 내 마음대로 구성 바꾸기

- 환경 변수 역시 IConfiguration을 통해 접근할 수 있도록 이미 기능이 내장되어있음
  - 예: Environment.GetEnvironmentVariable(“WINDIR”)과 builder.Configuration[“WINDIR”]이 동일한 결과를 반환
- 이 외에도 <프로그램 파일 이름>.<빌드 프로필>.JSON 같이 계층적으로 분리된 다중 JSON 파일 설정 지원이 내재됨

# 트리 형태의 구성값 참조

- JSON 자체가 트리 형태의 구성을 지닐 수 있으며, 환경 변수나 명령줄 스위치로 이런 값을 쉽게 표현 가능
  - 특히 이런 기능은 컨테이너 환경이나 *NIX 환경에서 유용하게 사용 가능
- 기본 문법: ‘:’ 문자를 사용하여 상/하 관계를 나타내며, 대/소문자 구분은 없음
  - 예: Configuration[“Globalization:DefaultLanguage”]
- 위의 예시를 환경 변수에서는 ‘:’ 문자 대신 ‘__‘ 로 대체 표기
  - 예: GLOBALIZATION__DEFAULTLANGUAGE
- 명령줄 스위치의 경우에는 다음과 같이 표기
  - 예: --globalization:defaultlanguage=ko

# 실제 코드 동작 예시 살펴보기

# 마무리

# 현대적인 애플리케이션 아키텍처의 새로운 표준

단순한 콘솔 애플리케이션을 넘어선 엔터프라이즈급 구조 제공

Microsoft가 권장하는 공식적인 호스팅 패턴 채택

Host.CreateDefaultBuilder를 통한 표준화된 진입점

IHostedService 인터페이스로 일관된 수명주기 관리

# 개발자 생산성 극대화

StartAsync(), StopAsync() 메서드로 명확한 수명주기 제어

CancellationToken을 통한 안전한 작업 취소 지원

ILogger 인터페이스로 구조화된 로깅 시스템 통합

의존성 주입으로 느슨한 결합과 테스트 용이성 확보

# 강력한 설정 관리

JSON, 환경 변수, 커맨드 라인 인자의 통합 관리

개발/운영 환경별 설정 분리 (appsettings.json)

User Secrets를 통한 개발 환경 설정 보안

# 미래 지향적 확장성

- 서비스 수명주기(Singleton, Scoped, Transient) 세밀한 제어
- 크로스 플랫폼 지원으로 다양한 환경 배포 가능
- 마이크로서비스 아키텍처로의 자연스러운 전환 지원

# 고맙습니다!

*linkedin.com/in/rkttu*