Dependency Injection in ASP.NET Core 6

Registering and Injecting Services

Steve Gordon

.NET Engineer and Microsoft MVP

@stevejgordon   www.stevejgordon.co.uk

Overview

Improving code with dependency injection

- Identify design problems
- Refactor to introduce abstractions
- Support dependency injection
- Register services with the container
- Inject framework dependencies
- Review the benefits

Later in This Course

The Microsoft dependency injection container

Registering more complex services

Injecting and resolving dependencies

Moving beyond the built-in container

Course Prerequisites

Fundamental knowledge and experience of .NET and ASP.NET Core

Experience programming in C#

Version Check

Version Check

This version was created by using:

- .NET 6.0
- C# 10
- Visual Studio 2022

Version Check

This course is 99% applicable to:

- .NET Core 3.1
- .NET 5.0
- Visual Studio 2013 to 2019
- Future .NET versions

Not Applicable

This course is NOT applicable to:

- .NET Framework 4.80 and earlier
- .NET Core 1.0
- .NET Core 1.1
- .NET Core 2.0
- .NET Core 2.1
- .NET Core 3.0

Relevant Notes

A note on frameworks and libraries:

- A new version of .NET releases each year
- The dependency injection library changes very little between versions
- Microsoft aim to maintain backward compatibility between releases

Follow Along

Follow along: Download the exercise files

The solution requires the latest .NET 6.0.x SDK
http://dot.net

An IDE such as Visual Studio Community Edition or an editor such as Visual Studio Code

Let’s Get Started

Introducing the Tennis Booking Application

Why Use Dependency Injection?

Dependency Injection

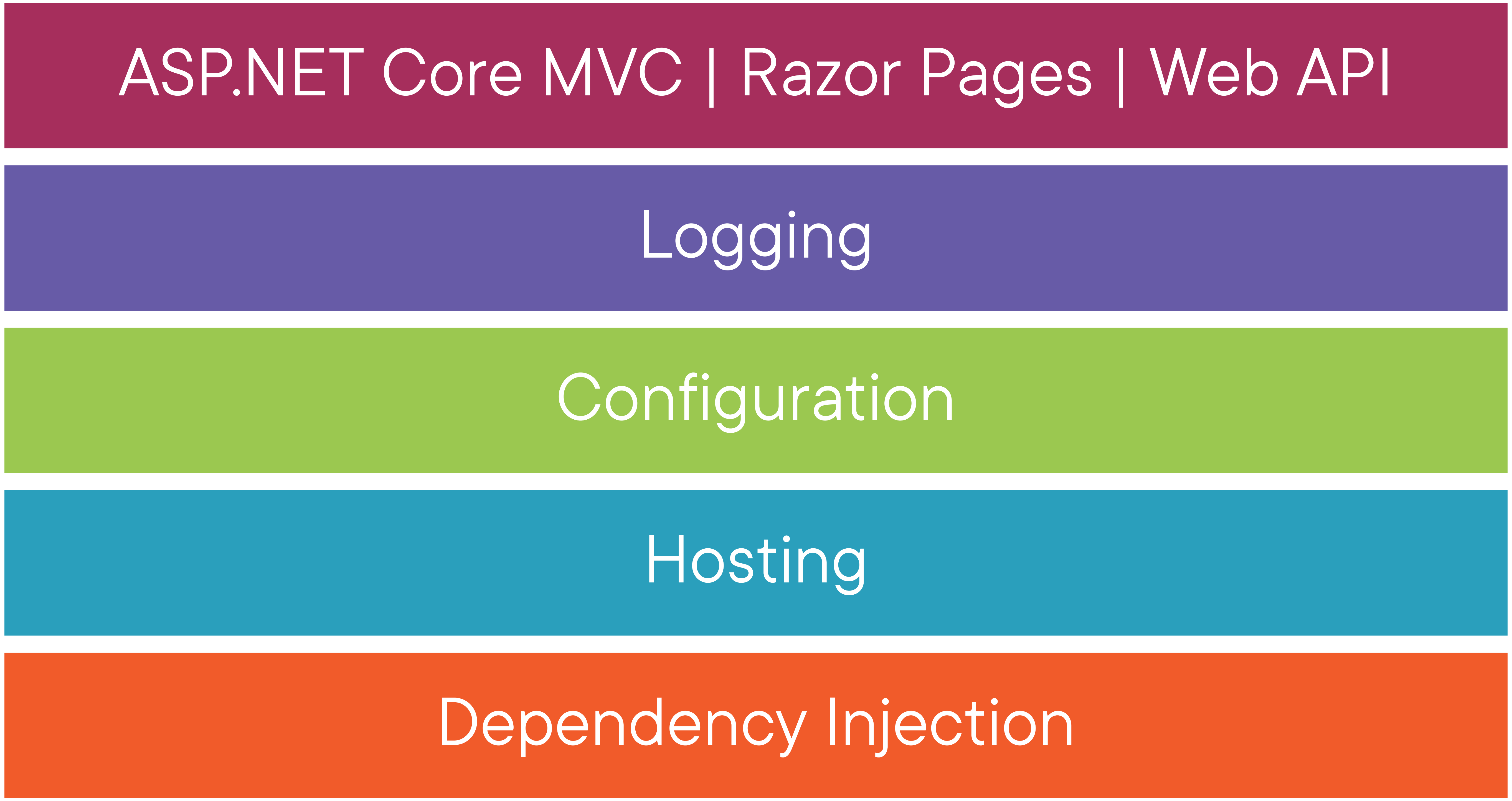
Dependency Injection

Patterns and Principles

Inversion of Control

Dependency Inversion Principle

ASP.NET Core Architecture



Recommendation

It is strongly recommended that we use dependency injection in ASP.NET Core applications.

Demo

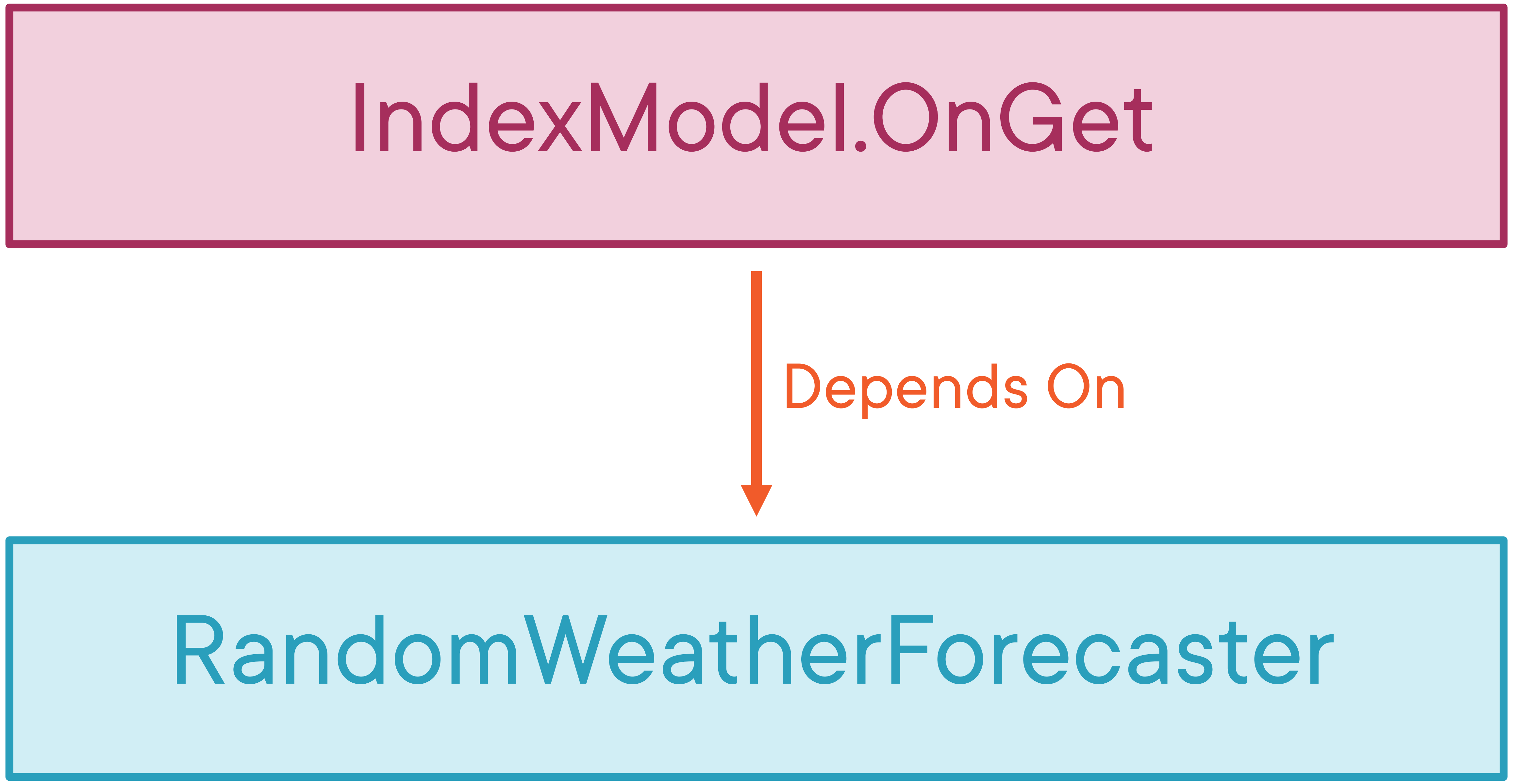
Identifying design problems in the Tennis Booking application

Class Dependencies

IndexModel.OnGet

RandomWeatherForecaster

Class Dependencies



Tight Coupling

The OnGet method is highly dependant on the RandomWeatherForecaster implementation

- It is therefore tightly coupled

Tight coupling is considered an anti-pattern

- Code is harder to maintain over time

Dependencies are normal

- We want to avoid direct coupling of classes

Tightly coupled code is harder to maintain. New requirements may require many classes to be updated.

Testing Challenges

The OnGet method is responsible for creating its dependency

- We can not control the implementation details nor the return value of the dependency during testing

Reliably testing the dependant class is tricky

Coding to Interfaces

Plan of Attack

Clean up code

Invert control

Apply dependency injection

“High-level modules should not depend on low-level modules. Both should depend on abstractions.”

“Abstractions should not depend upon details. Details should depend upon abstractions.”

Agile Principles, Patterns, and Practices in C#
Robert C. Martin and Micah Martin

Demo

Refactor existing code

- Reduce coupling in code

Extract an interface

Inverting Control with Constructor Injection

With constructor injection we define the list of required dependencies as parameters of the constructor for a class.

Example.cs

```
public class Example
{


    public Example()
    {

    }

    public void DoSomething()
    {
        var someService = new SomeService();
        someService.DoStuff();
    }
}
```

Example.cs

```
public class Example
{


    public Example(ISomeService someService)
    {

    }

    public void DoSomething()
    {
        var someService = new SomeService();
        someService.DoStuff();
    }
}
```

Example.cs

```
public class Example
{
    private readonly ISomeService _someService;

    public Example(ISomeService someService)
    {
        _someService = someService;
    }

    public void DoSomething()
    {
        var someService = new SomeService();
        someService.DoStuff();
    }
}
```

Example.cs

```
public class Example
{
    private readonly ISomeService _someService;

    public Example(ISomeService someService)
    {
        _someService = someService;
    }

    public void DoSomething()
    {
        ~~var someService = new SomeService();~~
        _someService.DoStuff();
    }
}
```

Inversion of Control

Inversion of Control

Inversion of Control

Inverting control of dependency creation

An external component creates dependencies

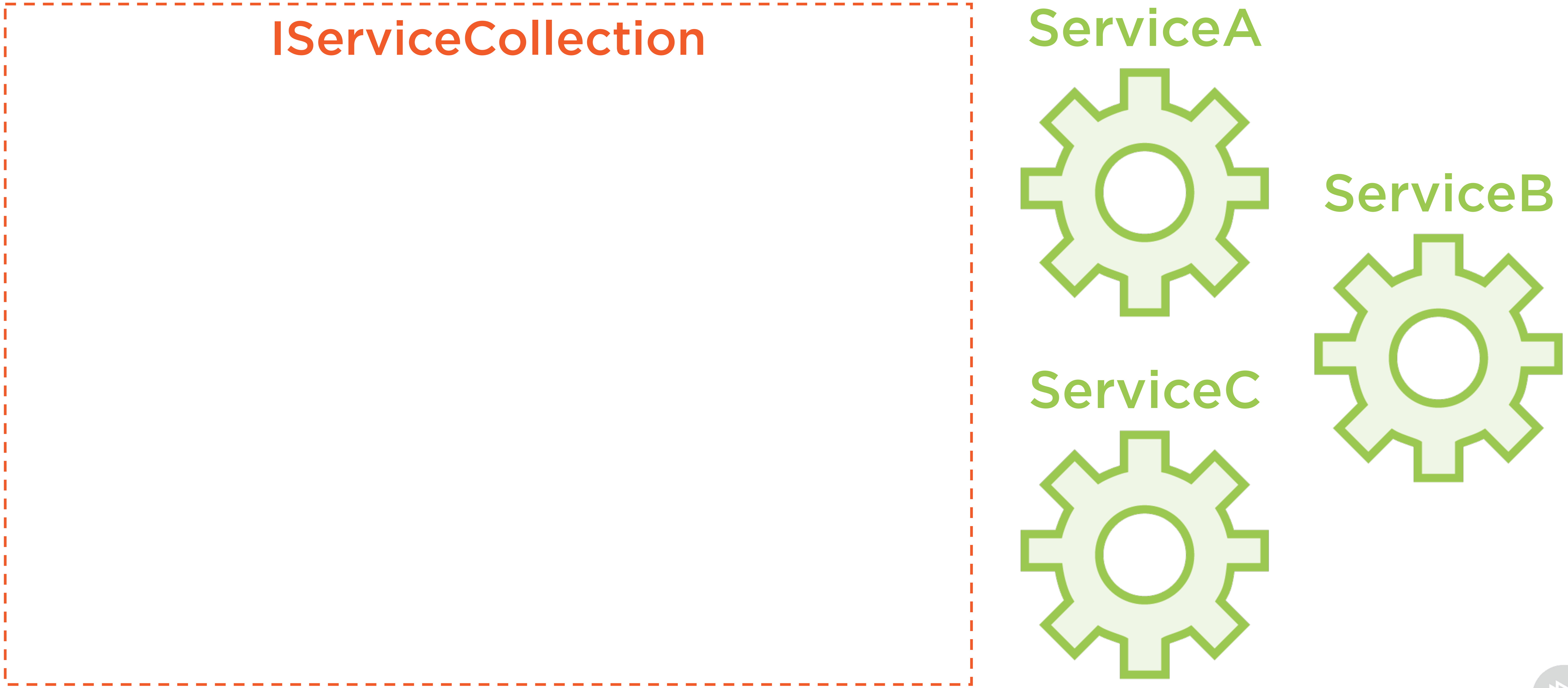
Combines with the dependency inversion principle to achieve loose coupling

Runtime Exception

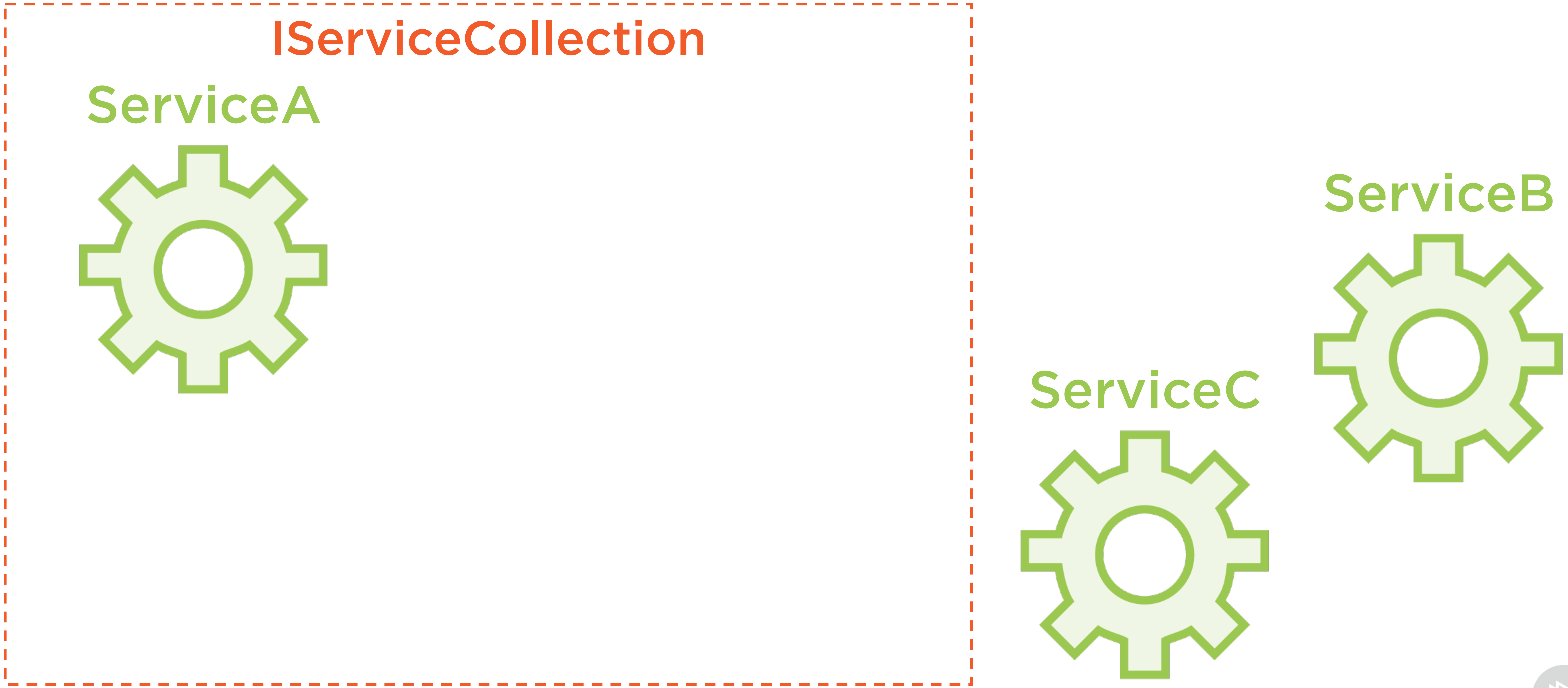
Missing or misconfigured service registrations may not be apparent until runtime when ASP.NET Core attempts to resolve them.

Registering Services

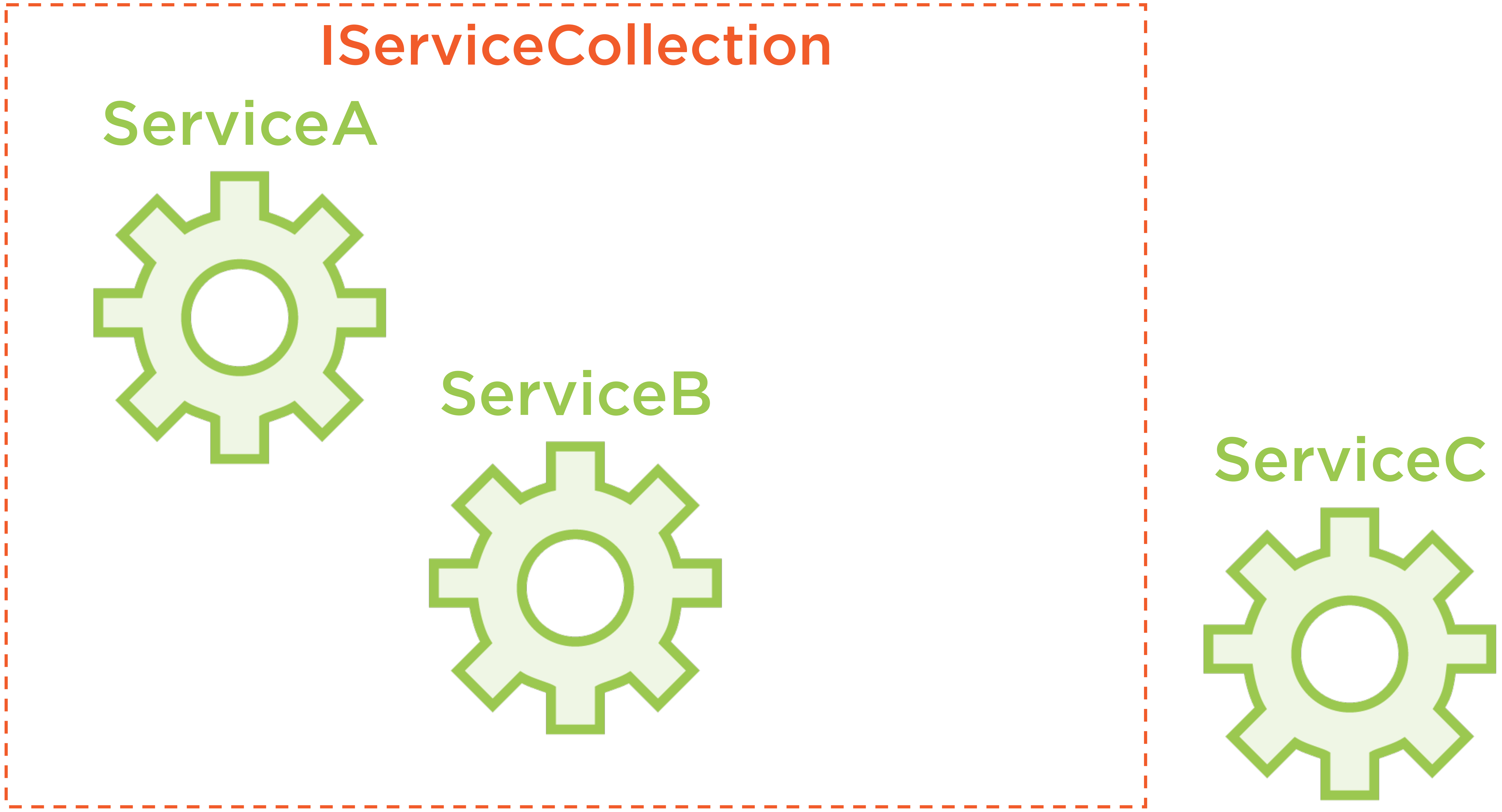
Registering Services



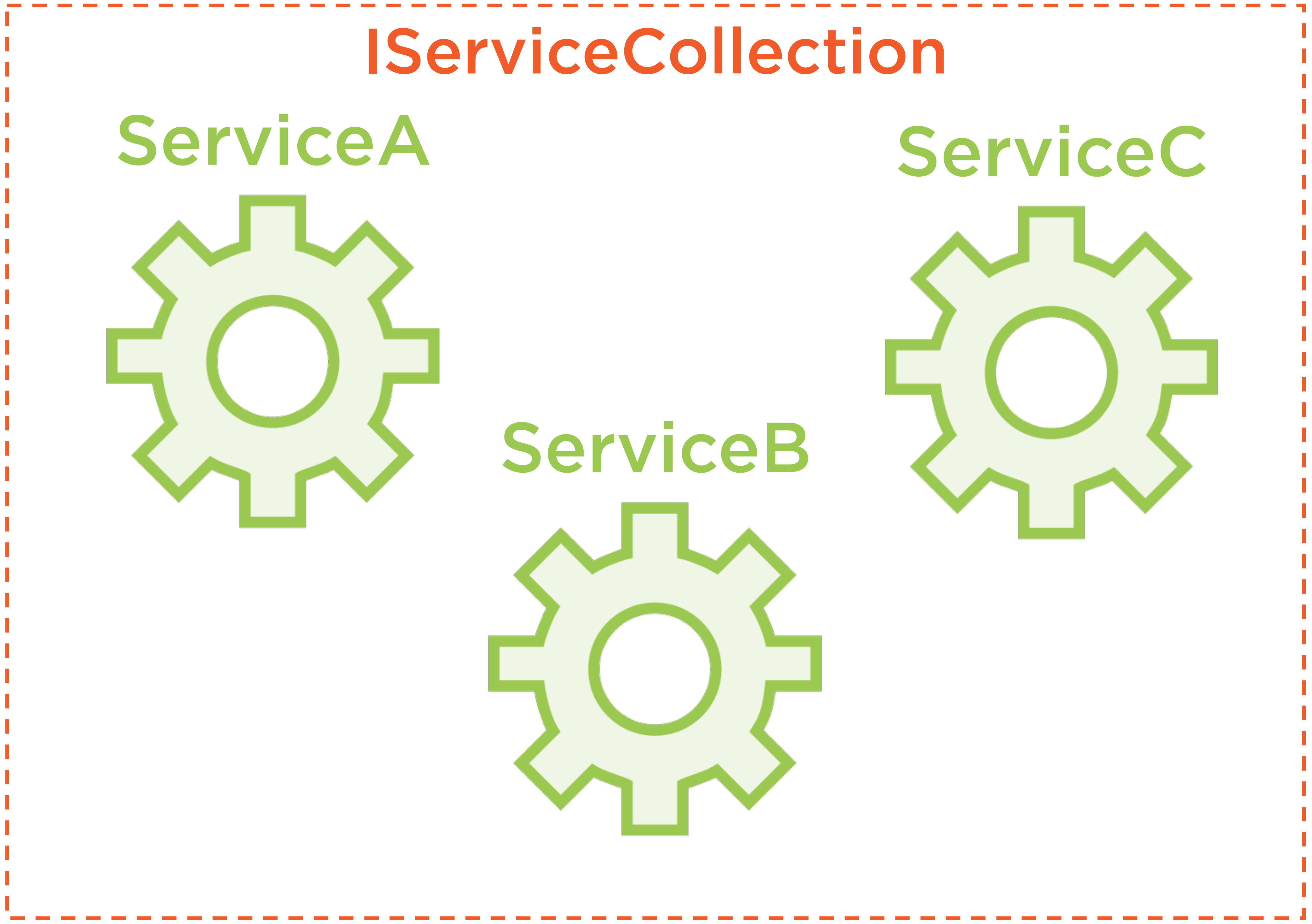
Registering Services



Registering Services



Registering Services



Register all required services (dependencies) with the IServiceCollection to avoid runtime exceptions.

Demo

Register our first service

ASP.NET Core 5 and earlier

Startup.cs

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // REGISTER SERVICES HERE

        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app)
    {
        // CONFIGURE REQUEST PIPELINE
    }
}
```

```
public static IServiceCollection AddTransient<TService,TImplementation>(this IServiceCollection services)
    where TService : class where TImplementation : class, Tservice
{
    // Implementation
}
```

AddTransient<TService, TImplementation>

Adds a transient service of the type specified in TService with an implementation type specified in TImplementation to the specified IServiceCollection.

The transient lifetime is a safe default until we learn more about service lifetimes.

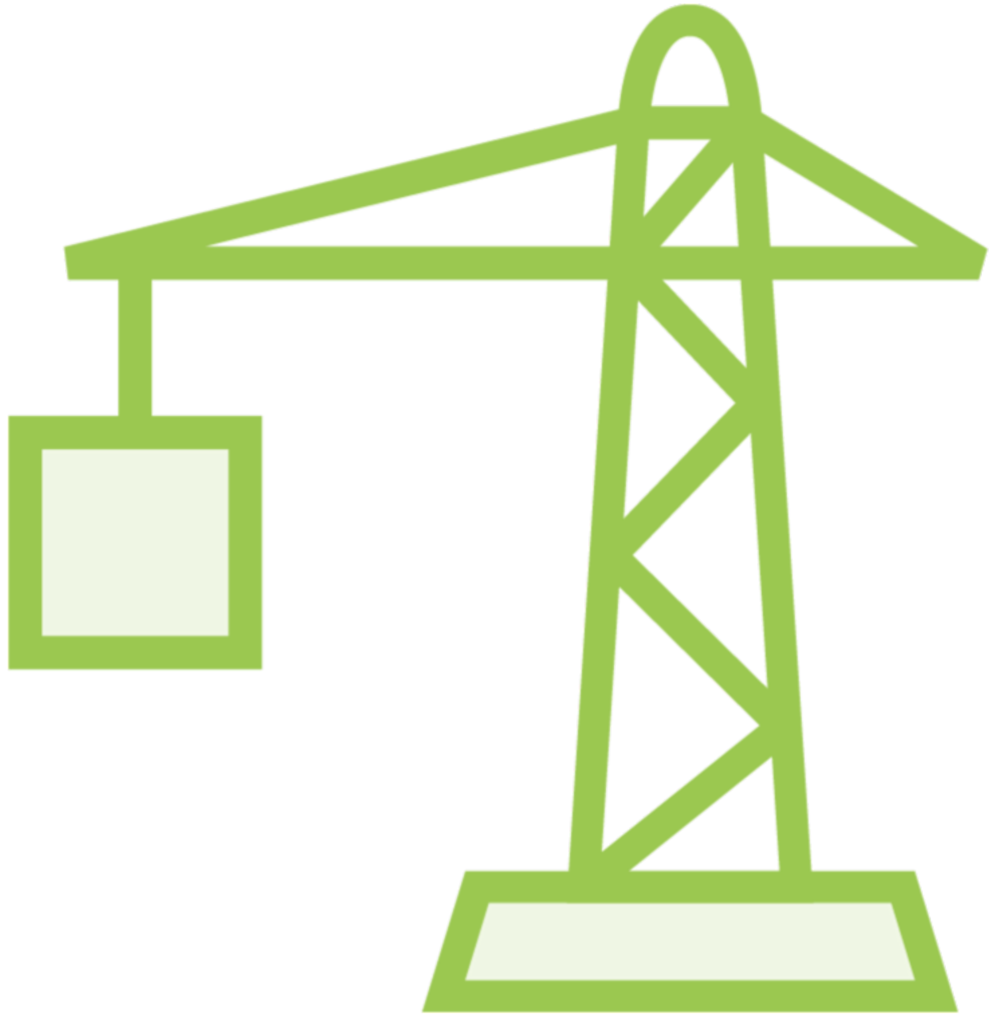
Order of Service Registration

Generally, services can be registered in any order

An exception to this is when intentionally registering multiple implementations of the same abstraction

Injecting Framework Dependencies

ASP.NET Core Framework Services



- Logging
- Configuration and options
- Application lifetime
- Hosting environment
- Various factories
- Startup filters
- Object pooling
- Routing

Demo

Inject a logger

Advantages of Dependency Injection

Advantages

Promotes loose coupling of components

Promotes logical abstraction of components

Supports unit testing

Cleaner, more readable code

Improved Testing

Can manually construct classes under test after applying inversion of control, providing fakes, mocks or stubs

After introducing abstractions and applying the dependency inversion principle, mocking dependencies during testing is made simpler

Up Next:

The Microsoft Dependency Injection Container