The Microsoft Dependency Injection Container

Steve Gordon

.NET Engineer and Microsoft MVP

@stevejgordon   www.stevejgordon.co.uk

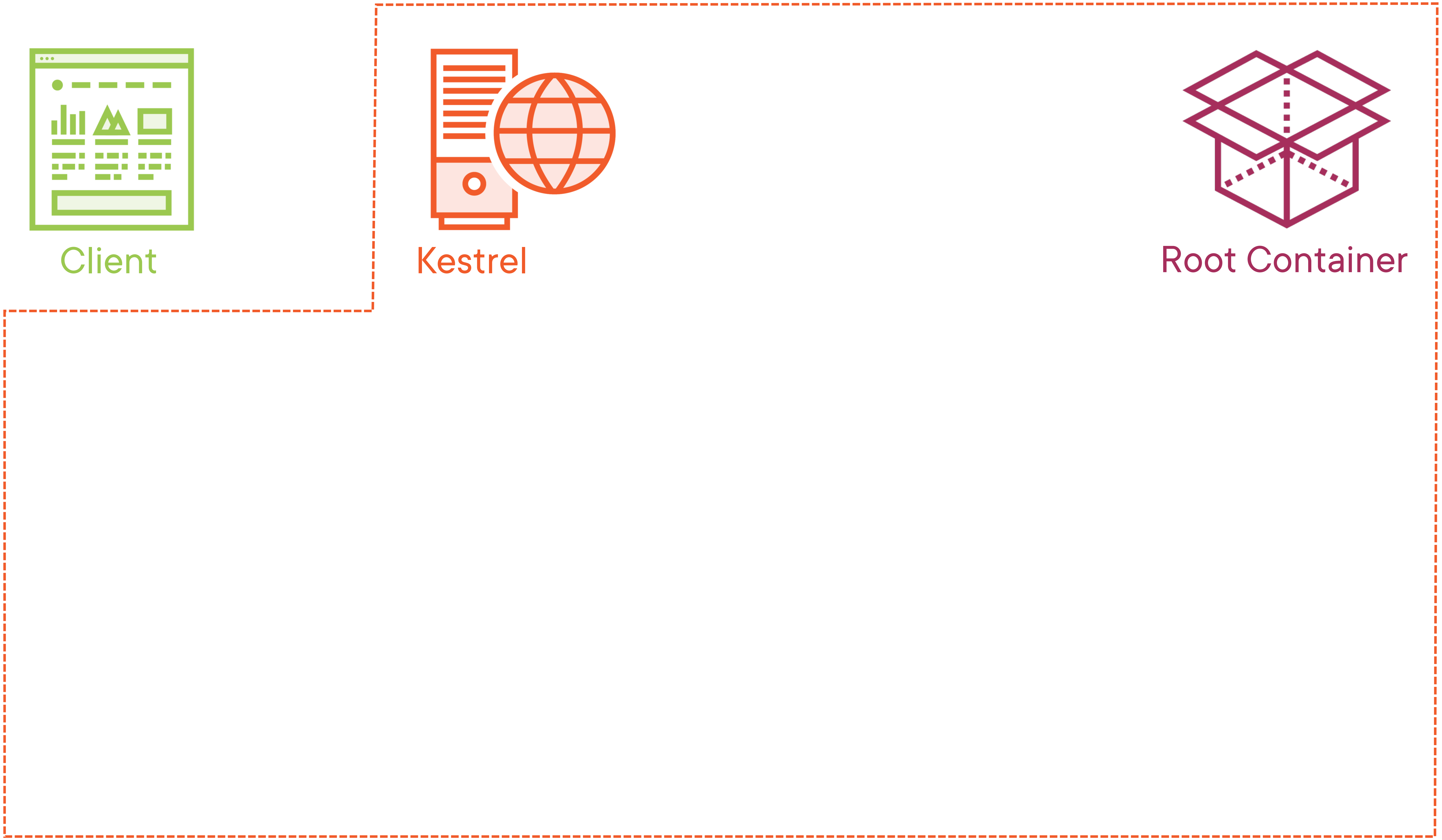
Overview

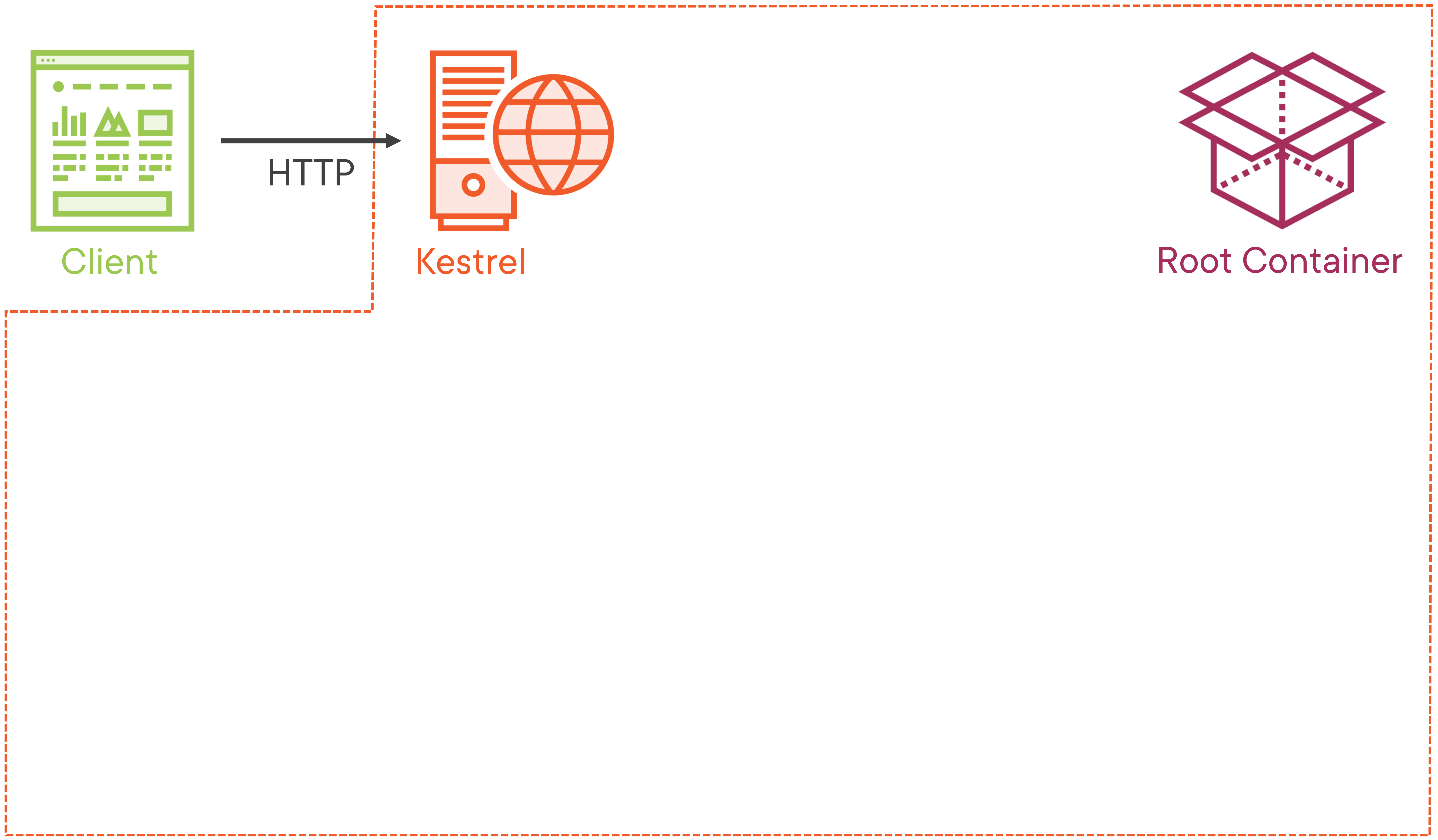
How ASP.NET Core uses the container

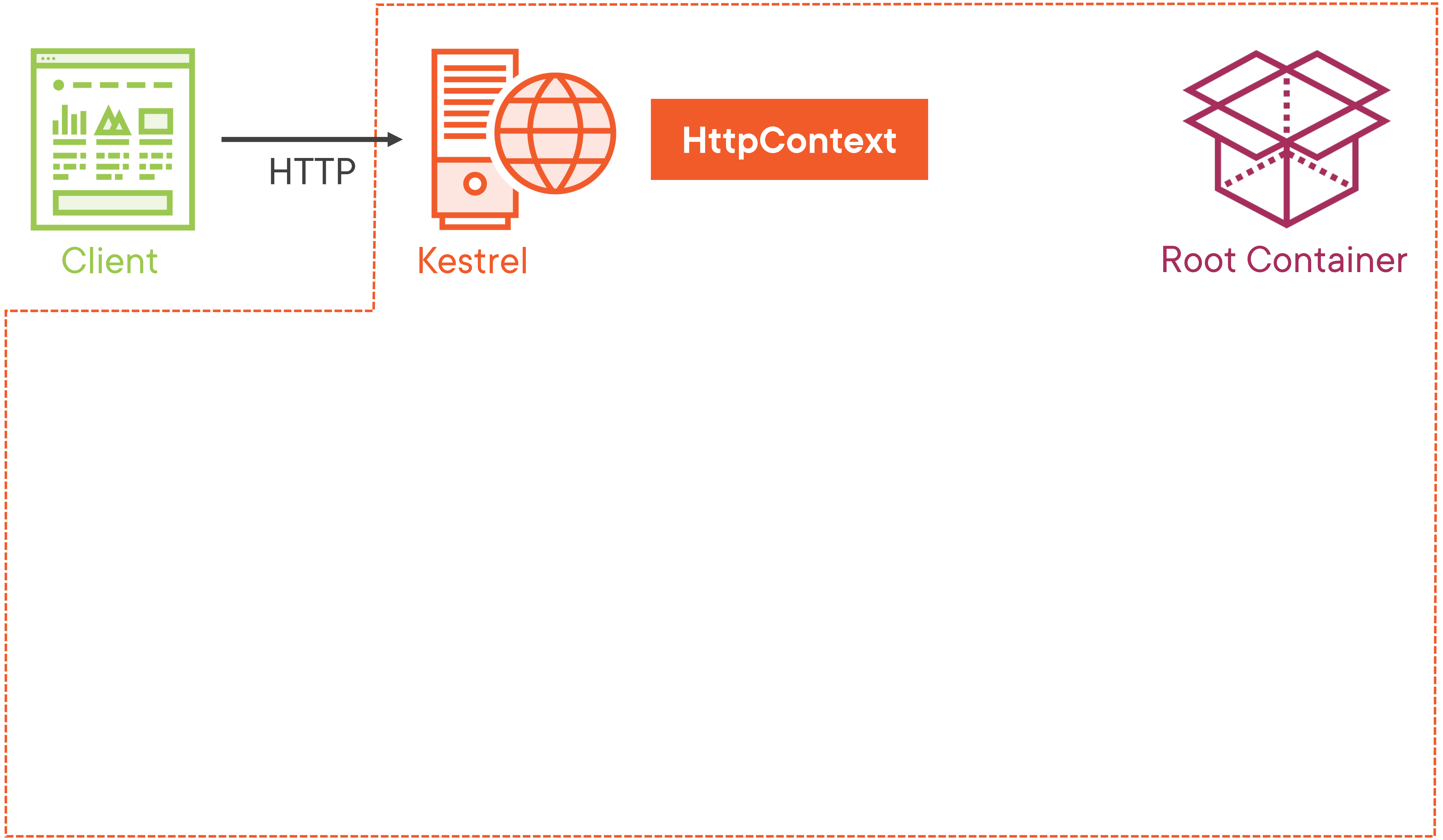
What to register with the D.I. container

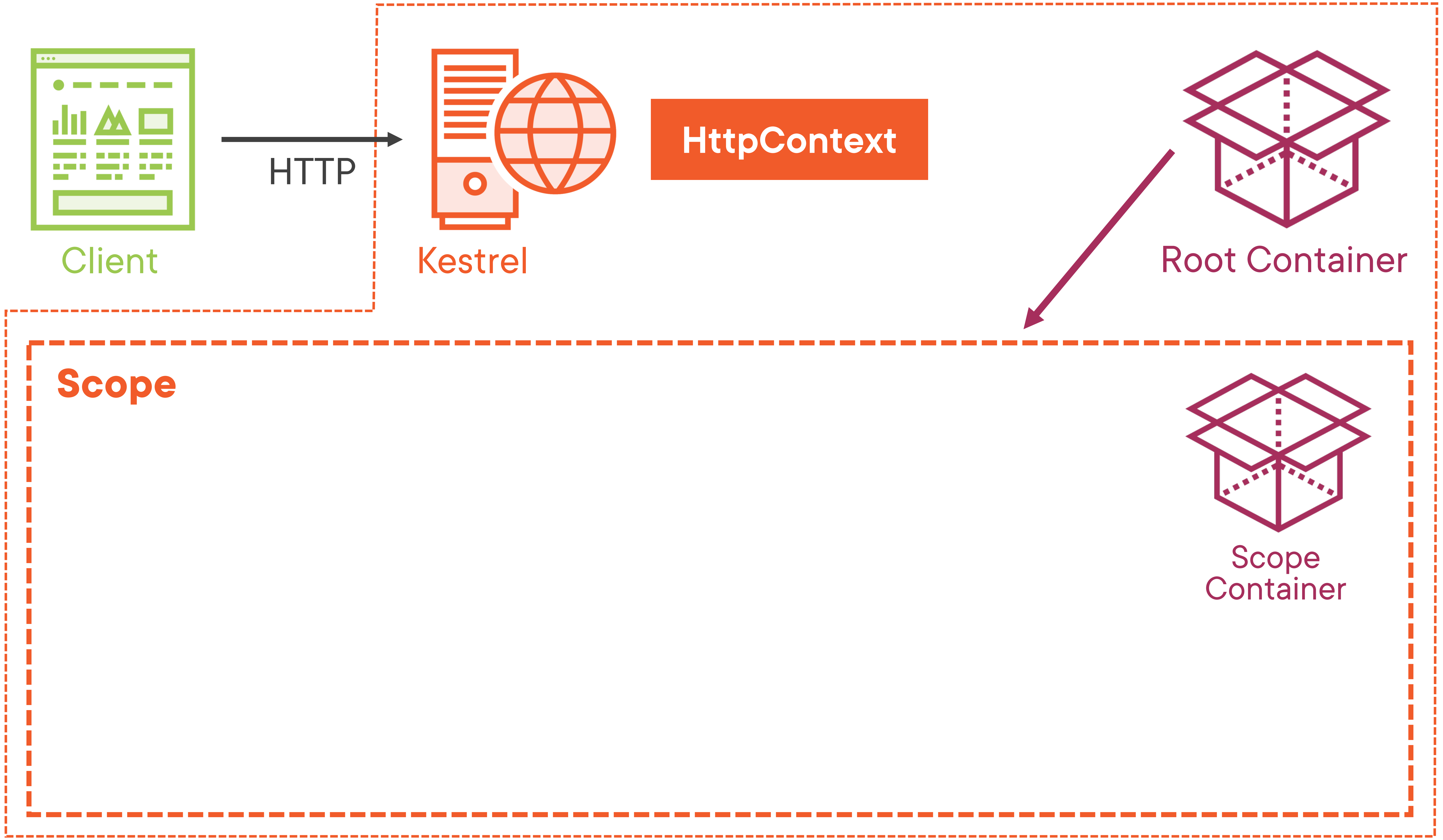
Service lifetimes

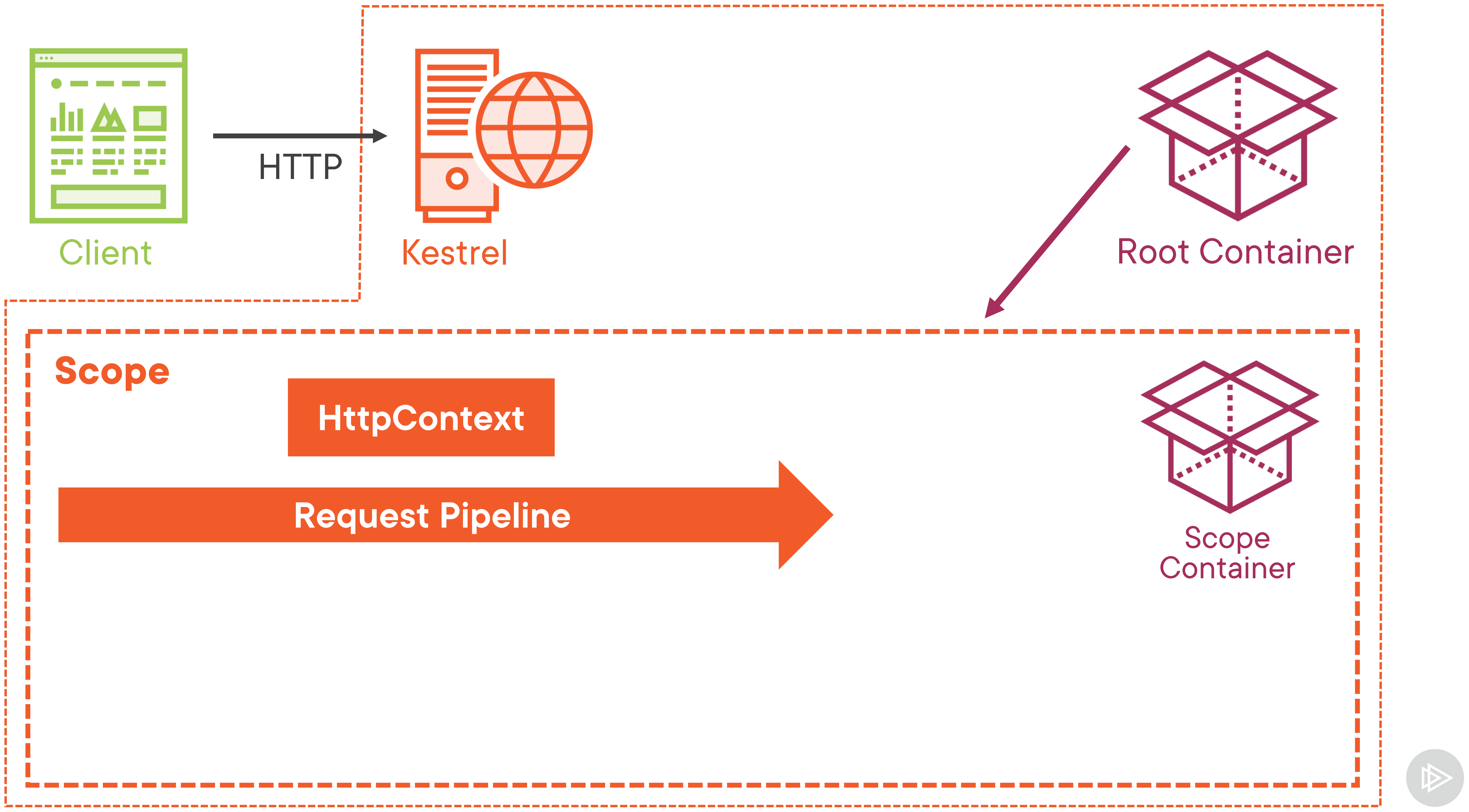
The ASP.NET Core Request Lifecycle

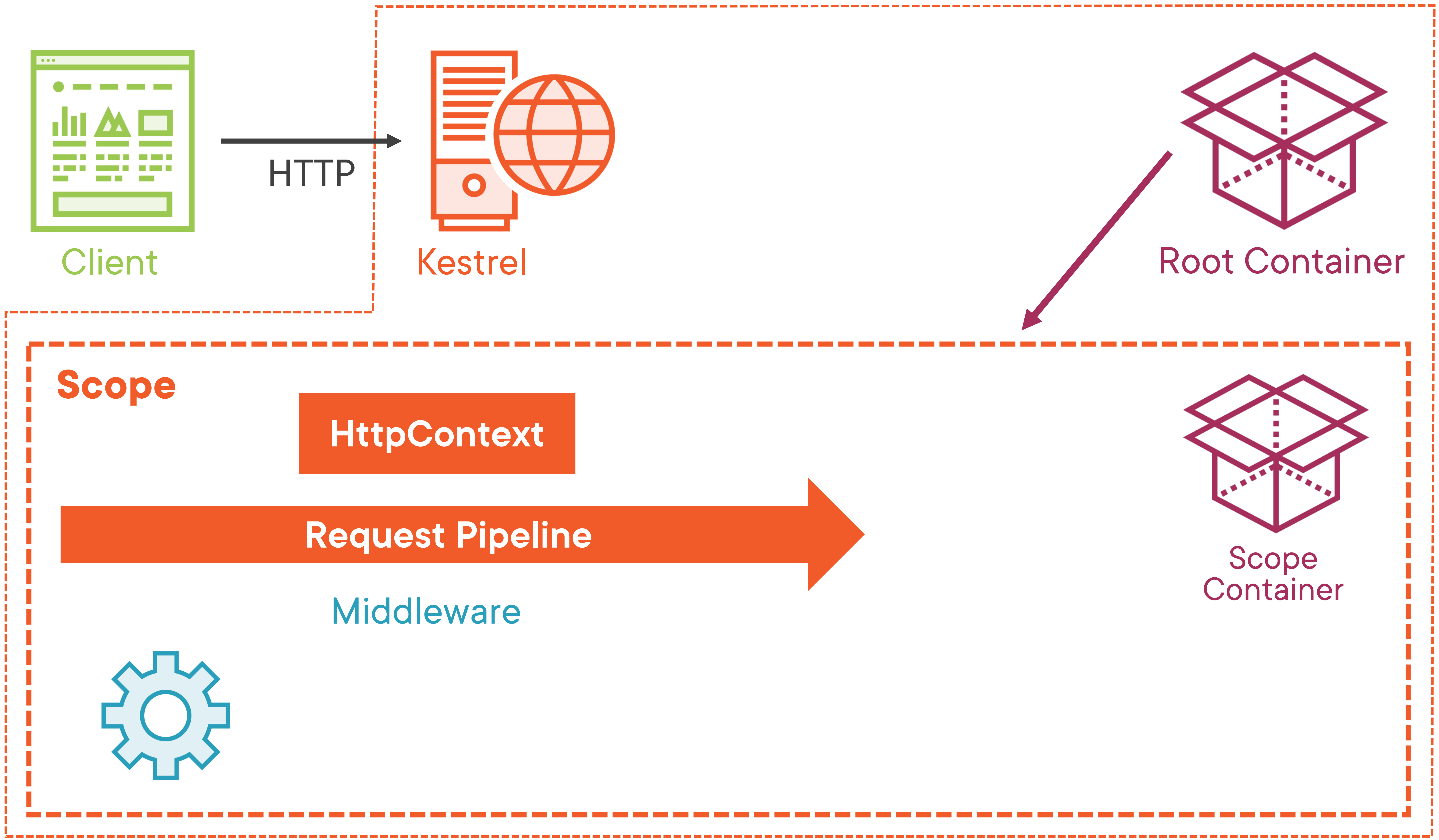

Client
Kestrel
Root Container

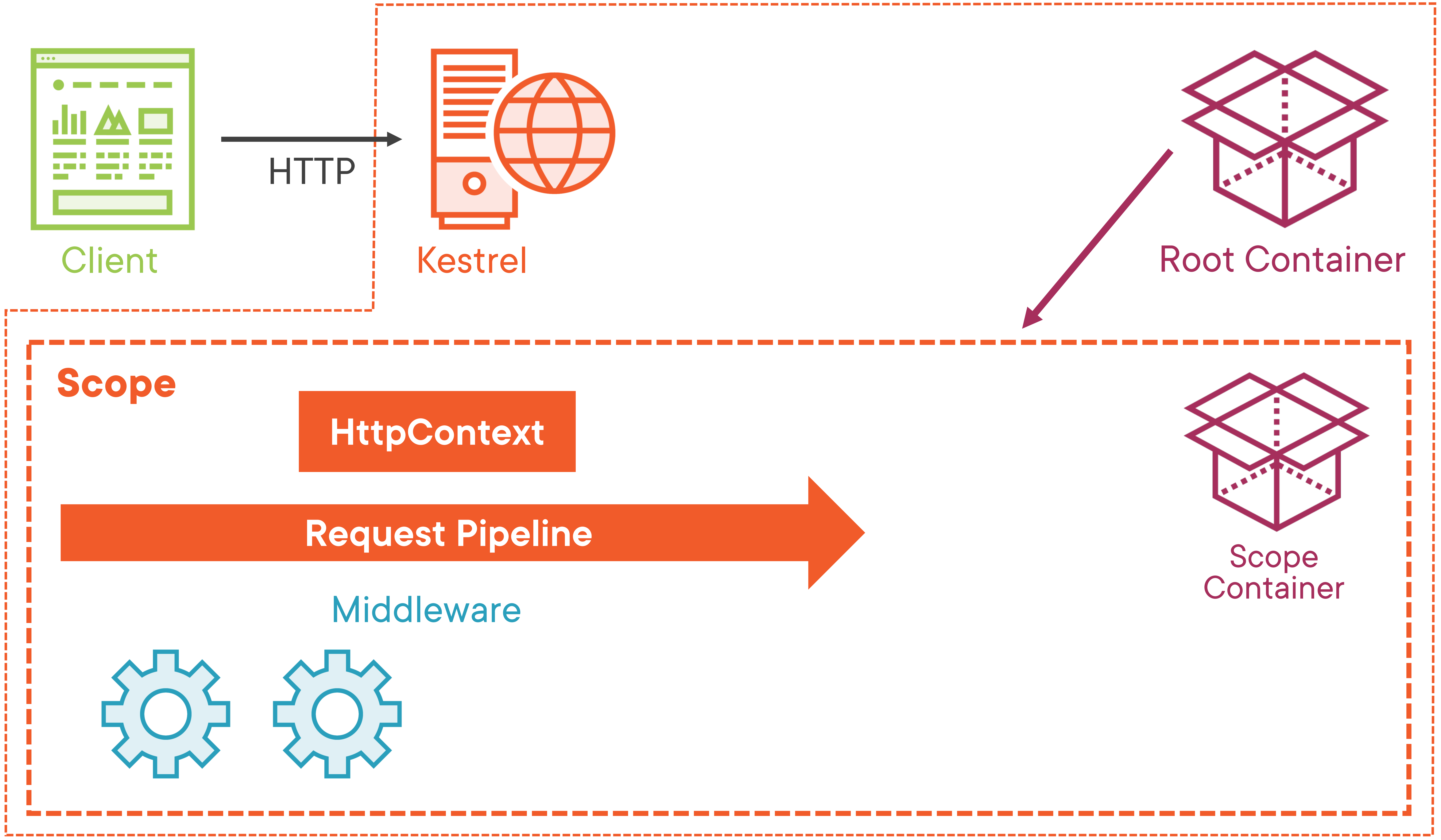

Client
HTTP
Kestrel
Root Container

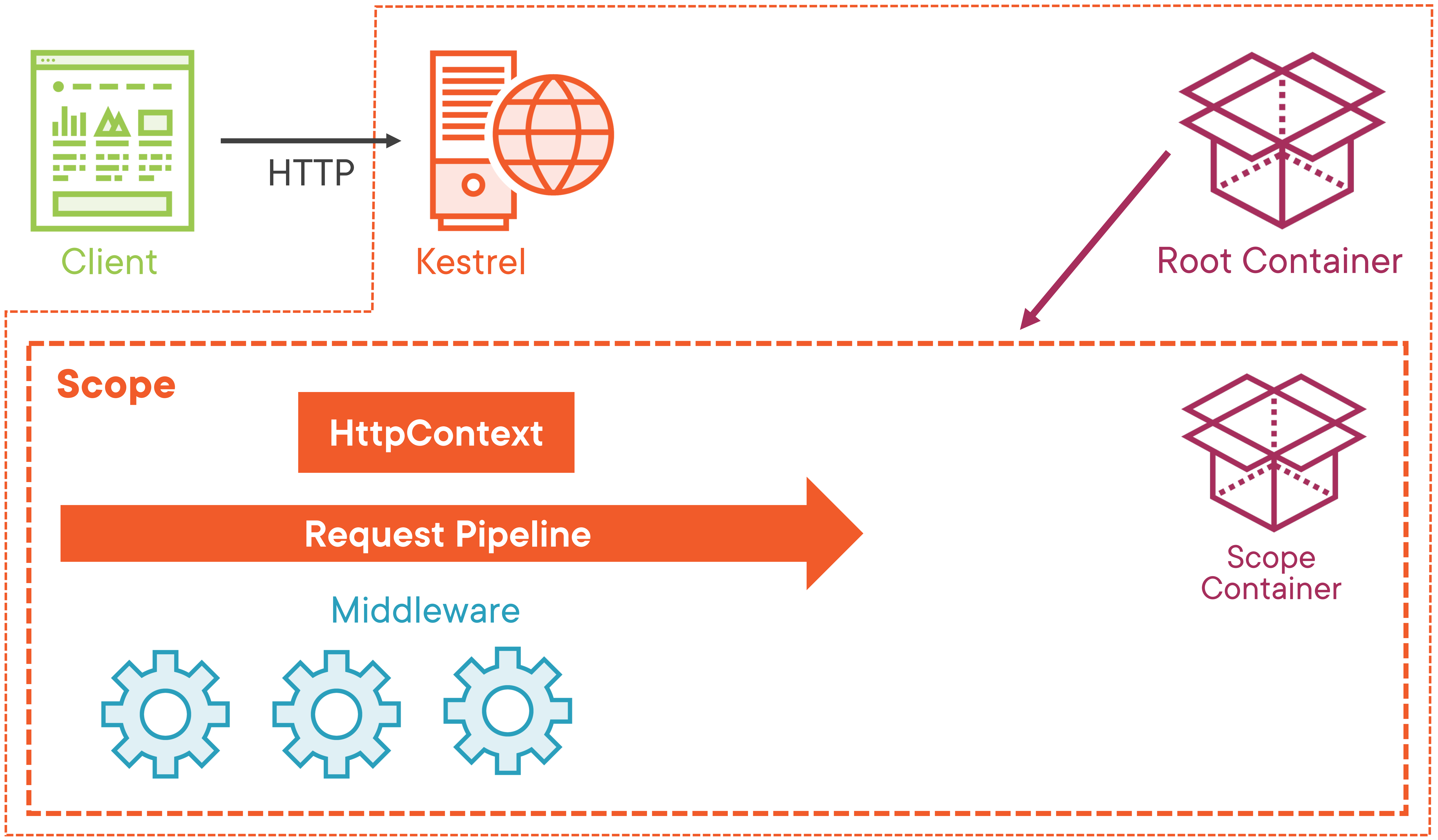

Client
HTTP
Kestrel
HttpContext
Root Container

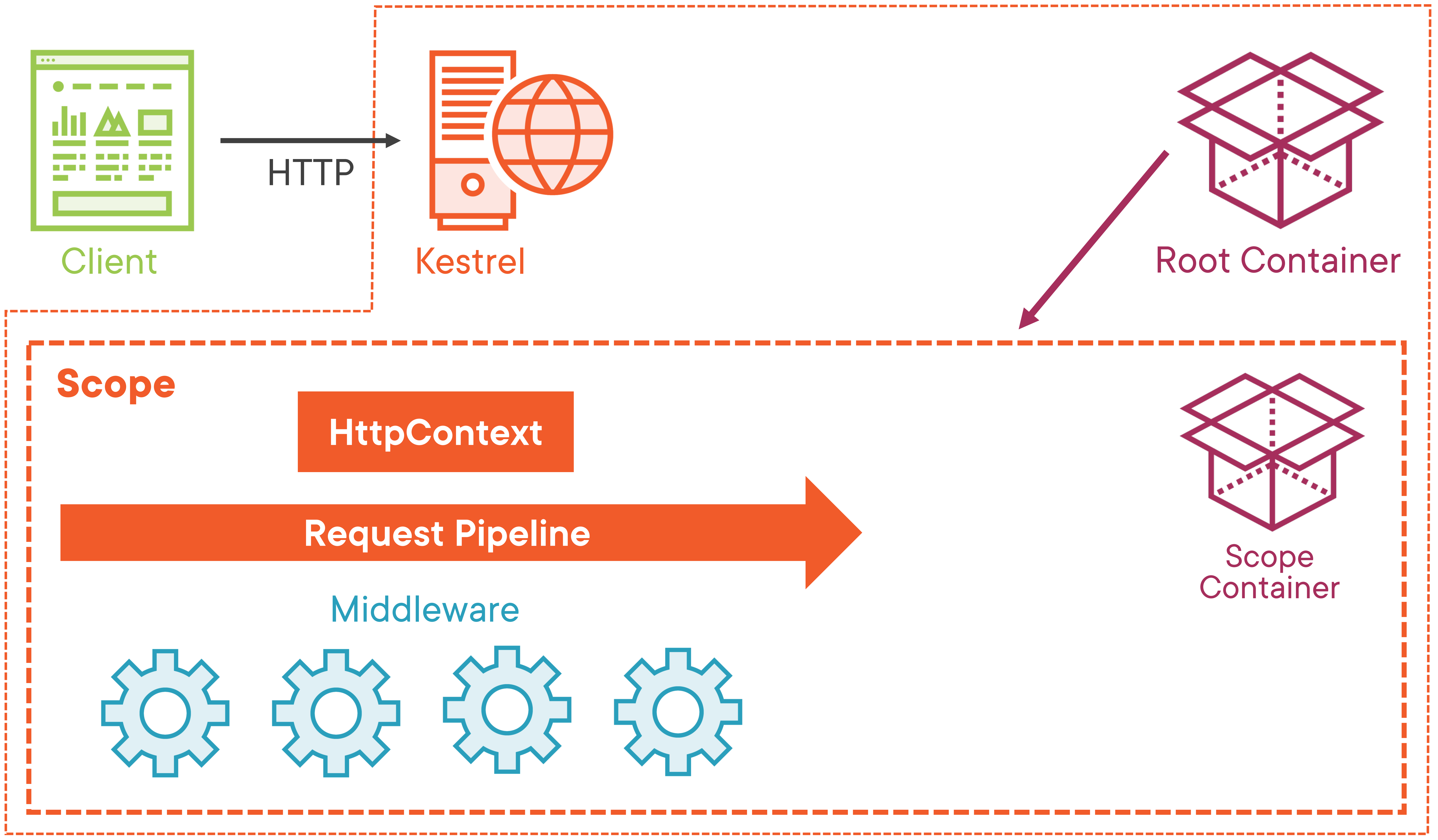

Client
HTTP
Kestrel
HttpContext
Root Container
Scope
Scope Container

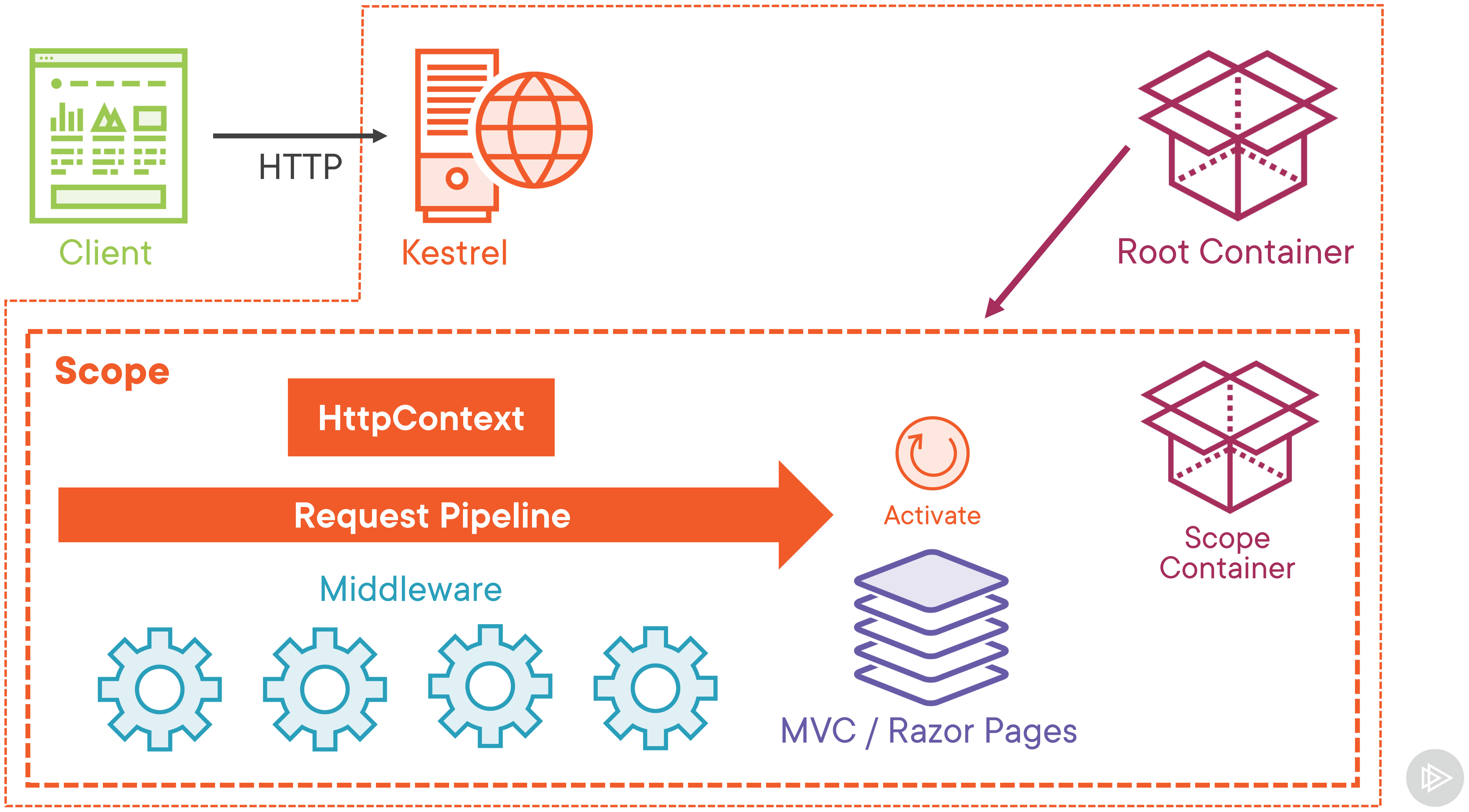

Client
HTTP
Kestrel
Root Container
Scope
HttpContext
Request Pipeline
Scope Container

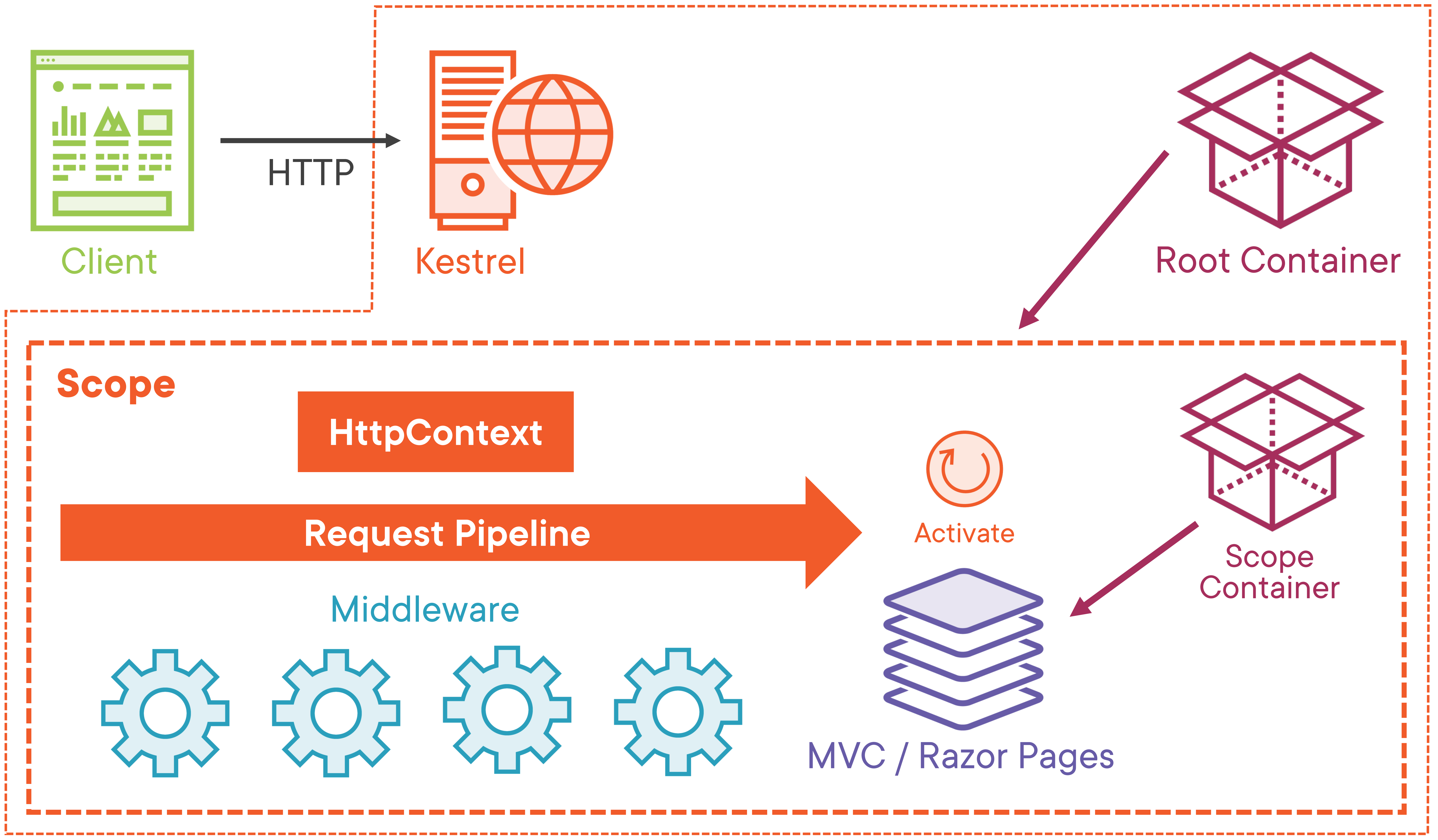

Client
HTTP
Kestrel
Root Container
Scope
HttpContext
Request Pipeline
Middleware
Scope Container


Client
HTTP
Kestrel
Root Container
Scope
HttpContext
Request Pipeline
Middleware
Scope Container

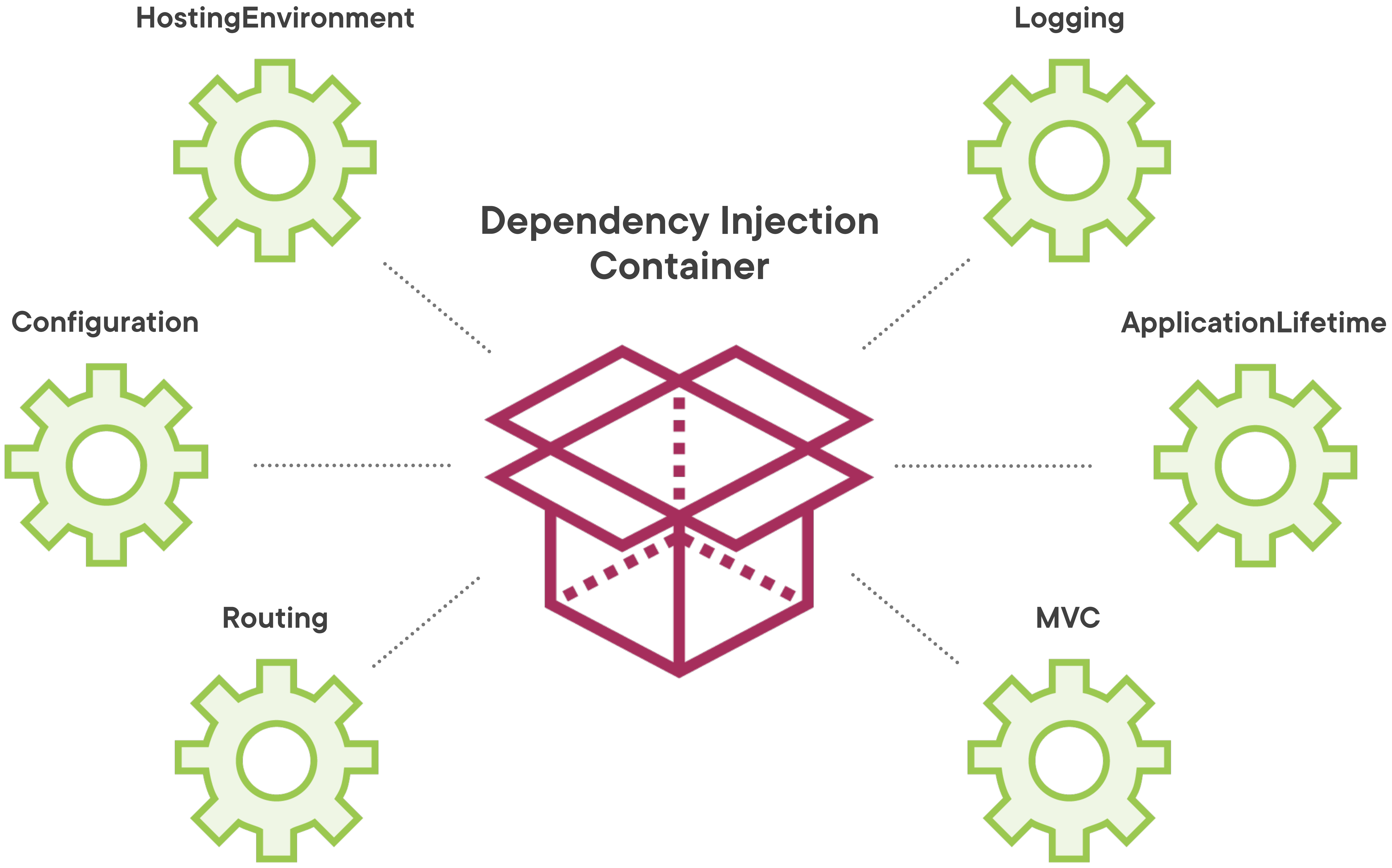

Client
HTTP
Kestrel
Root Container
Scope
HttpContext
Request Pipeline
Middleware
Scope Container


Client
HTTP
Kestrel
Root Container
Scope
HttpContext
Request Pipeline
Middleware
Scope Container

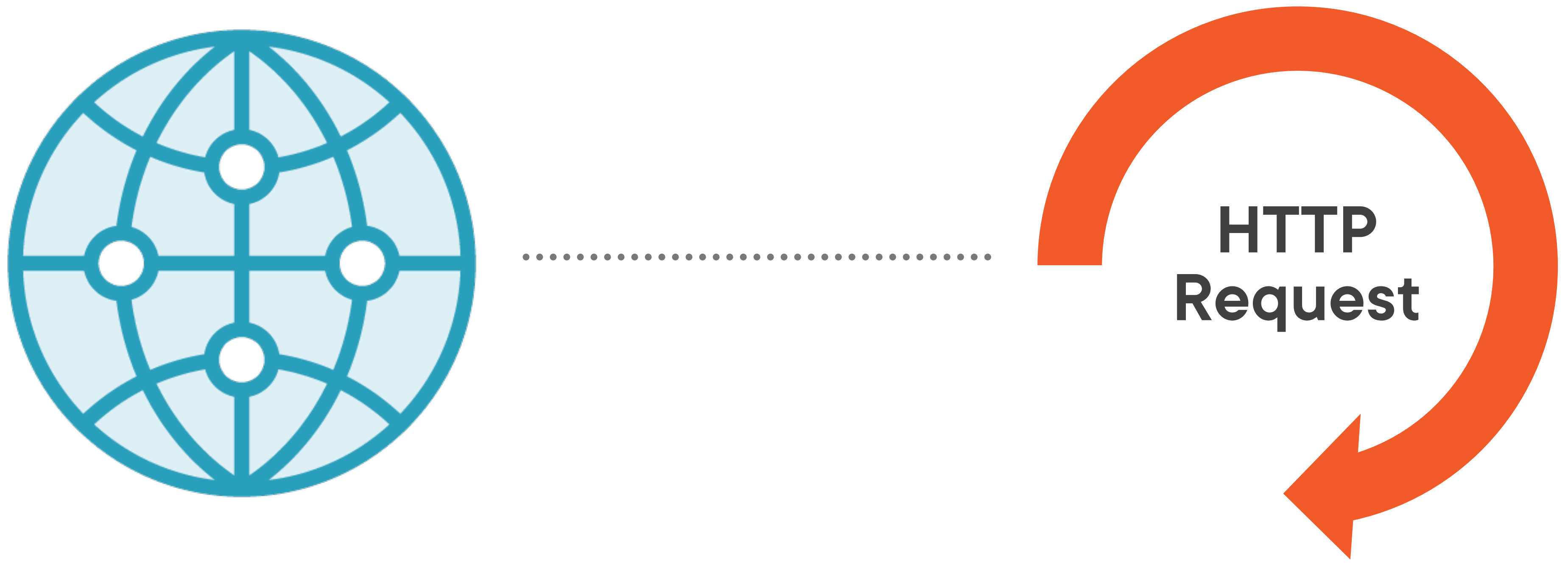

Client
HTTP
Kestrel
Root Container
Scope
HttpContext
Request Pipeline
Activate
Scope Container
Middleware
MVC / Razor Pages


Client
HTTP
Kestrel
Root Container
Scope
HttpContext
Request Pipeline
Activate
Scope Container
Middleware
MVC / Razor Pages

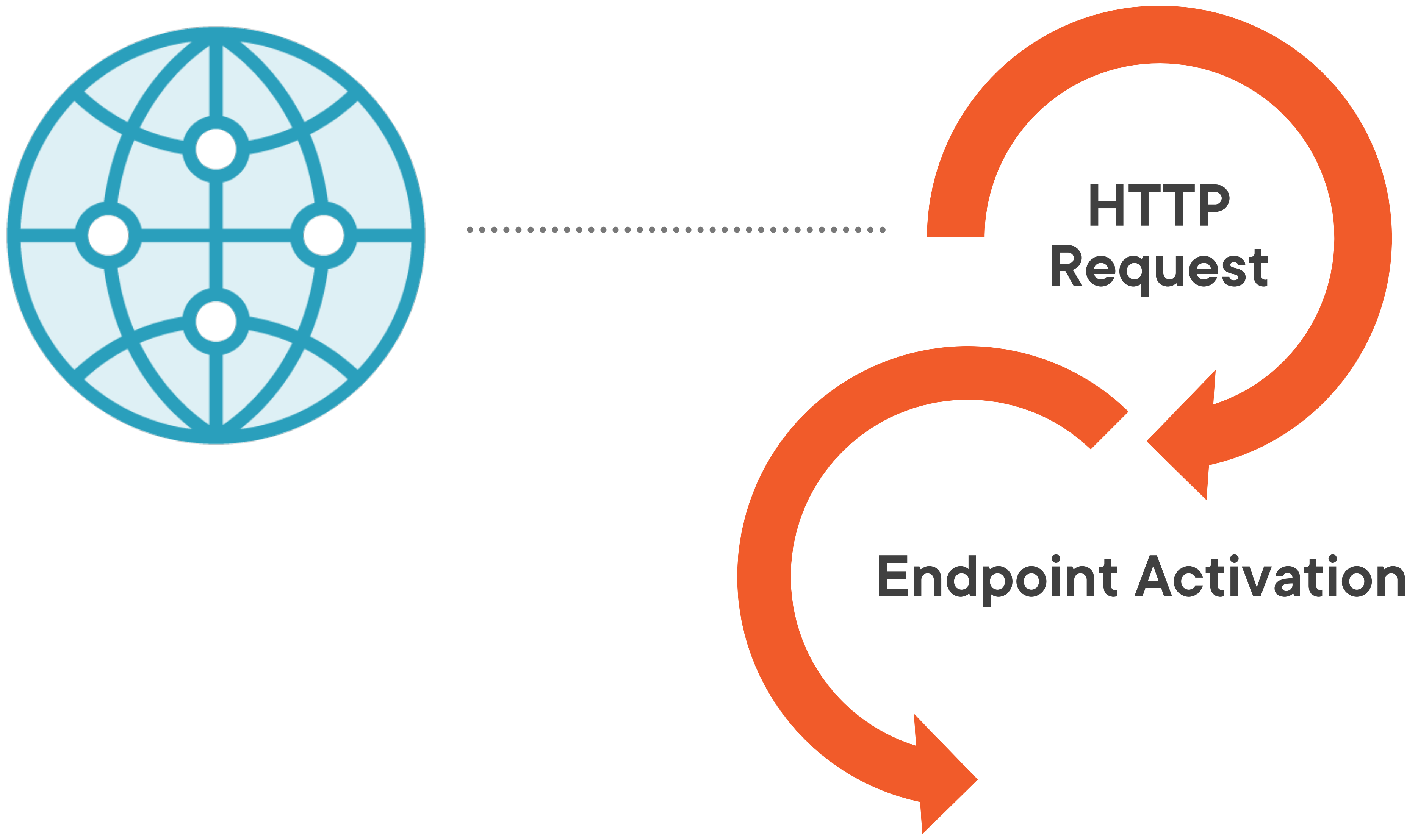
The Microsoft Dependency Injection Container


HostingEnvironment
Logging
Dependency Injection Container
Configuration
ApplicationLifetime
Routing
MVC

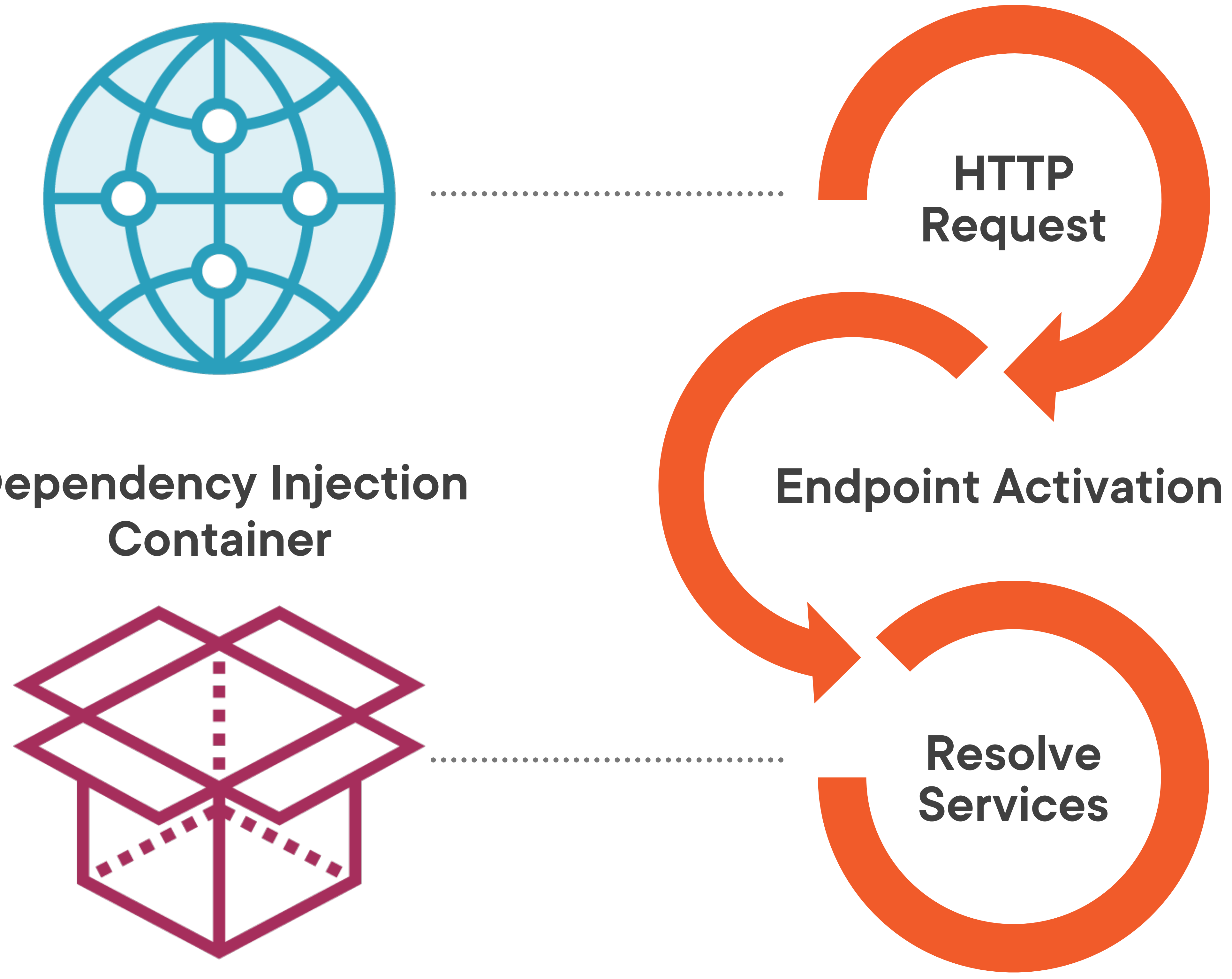
ASP.NET Core and Dependency Injection



ASP.NET Core and Dependency Injection



ASP.NET Core and Dependency Injection



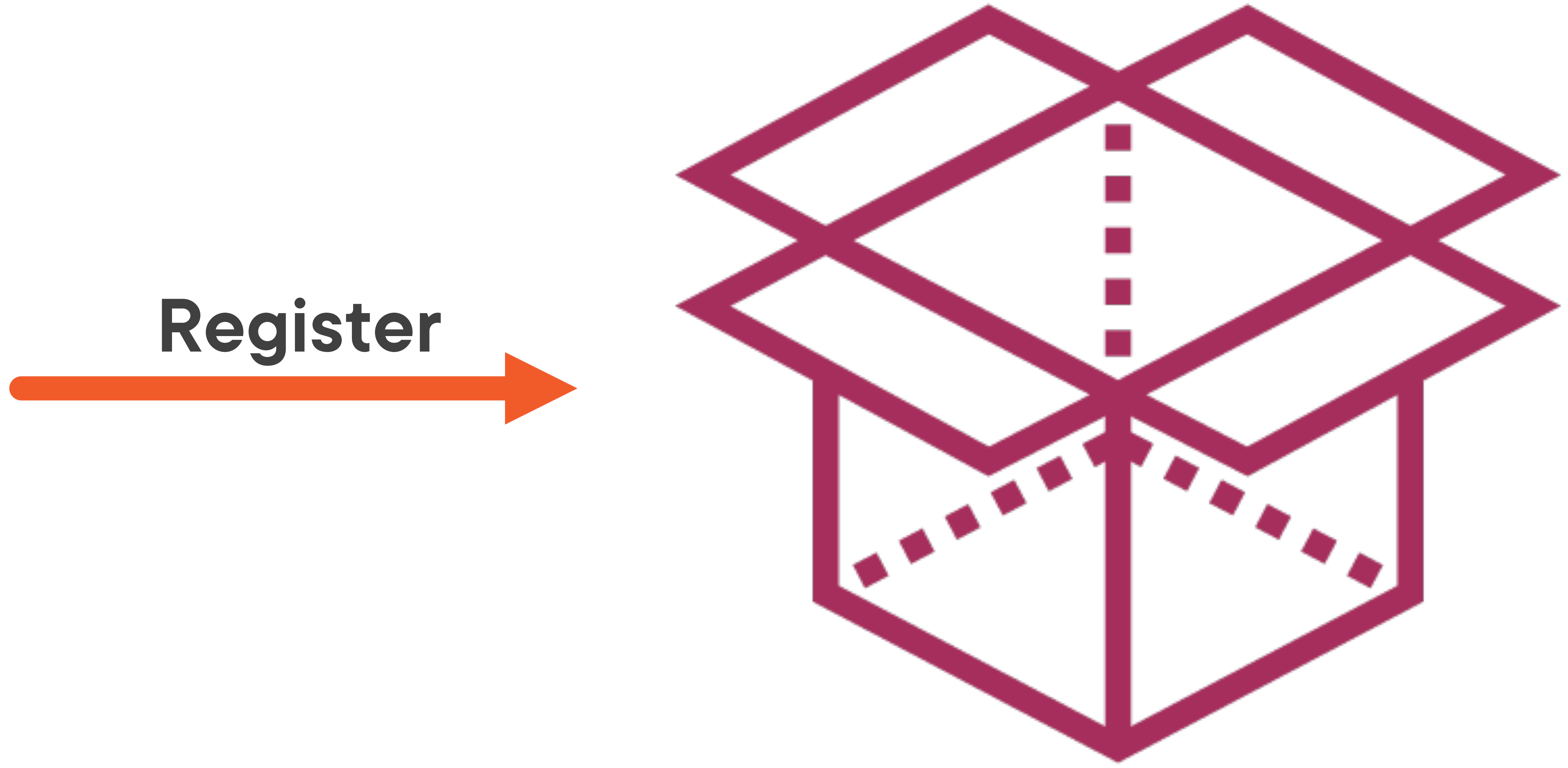
Microsoft.Extensions.DependencyInjection

Microsoft.AspNetCore.App

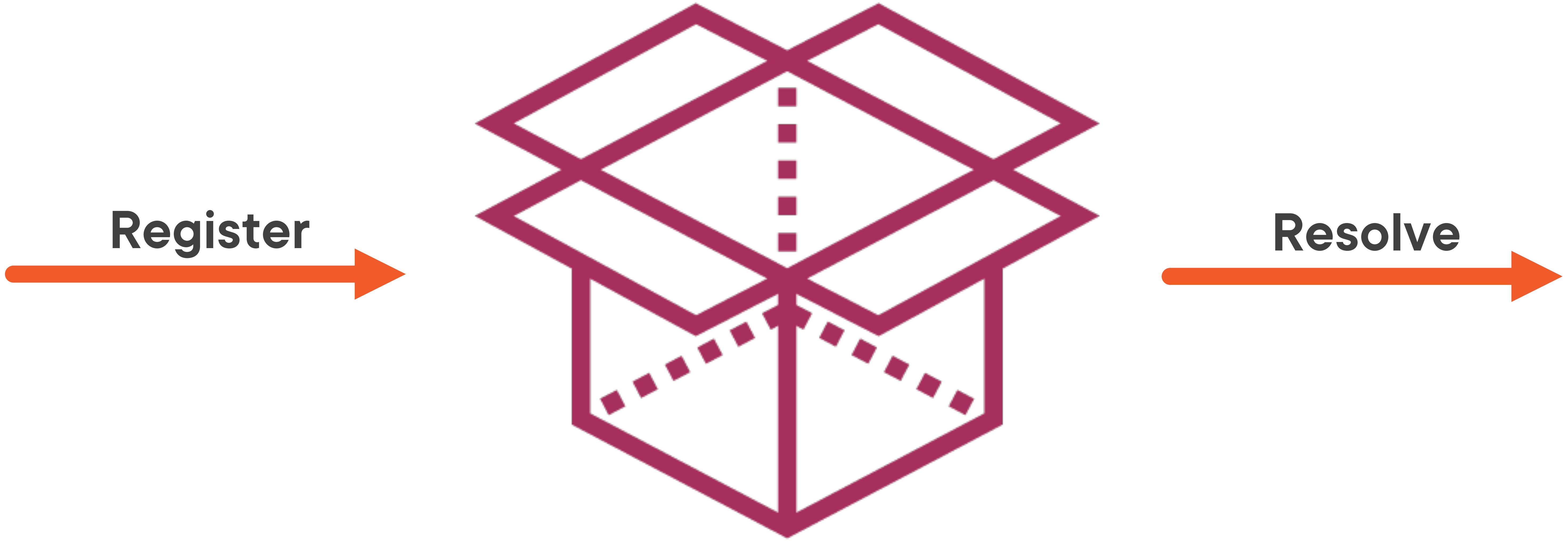
Dependency Injection
Container
=
Inversion of Control
Container

A dependency injection container is not a requirement to apply dependency injection. Using one simplifies management of dependencies.

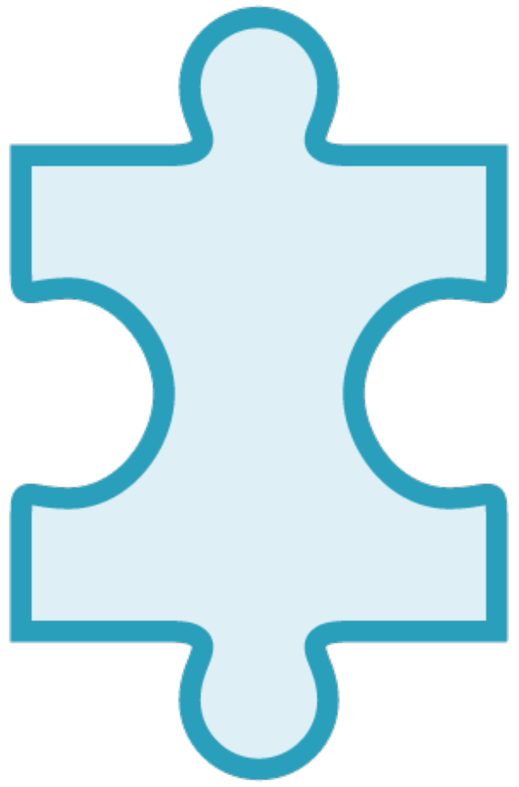
Dependency Injection Containers



Dependency Injection Containers

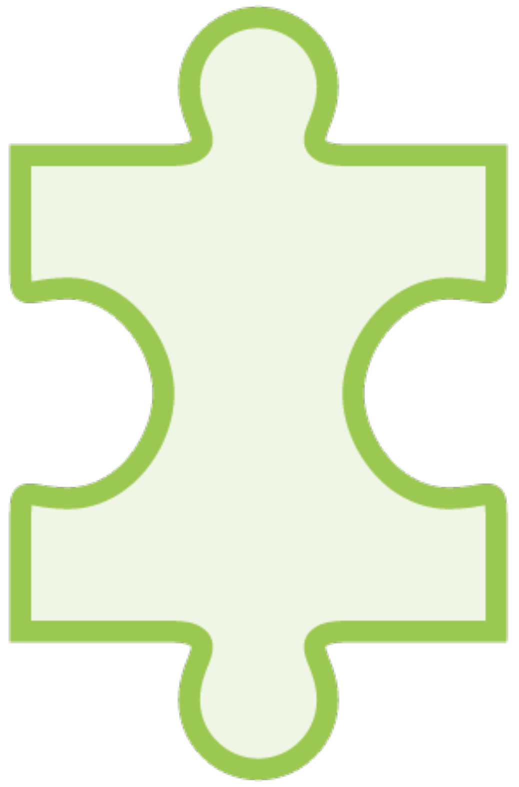


Components



IServiceCollection

Register services



IServiceProvider

Resolve service instances

What to Register with the D.I. Container

Identifying Dependencies

Locate ‘new’ keyword usage

Is the object a dependency?

- Are methods called on the type which are required for the consuming type to function?

Apply dependency inversion

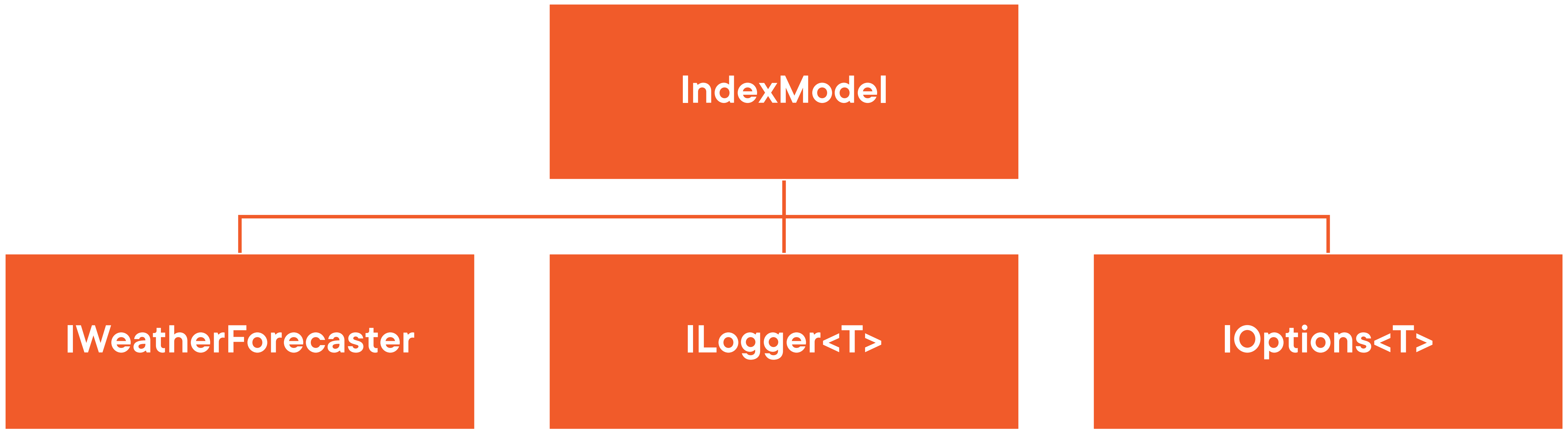
- Accept the dependency via the constructor

Register the service with the container

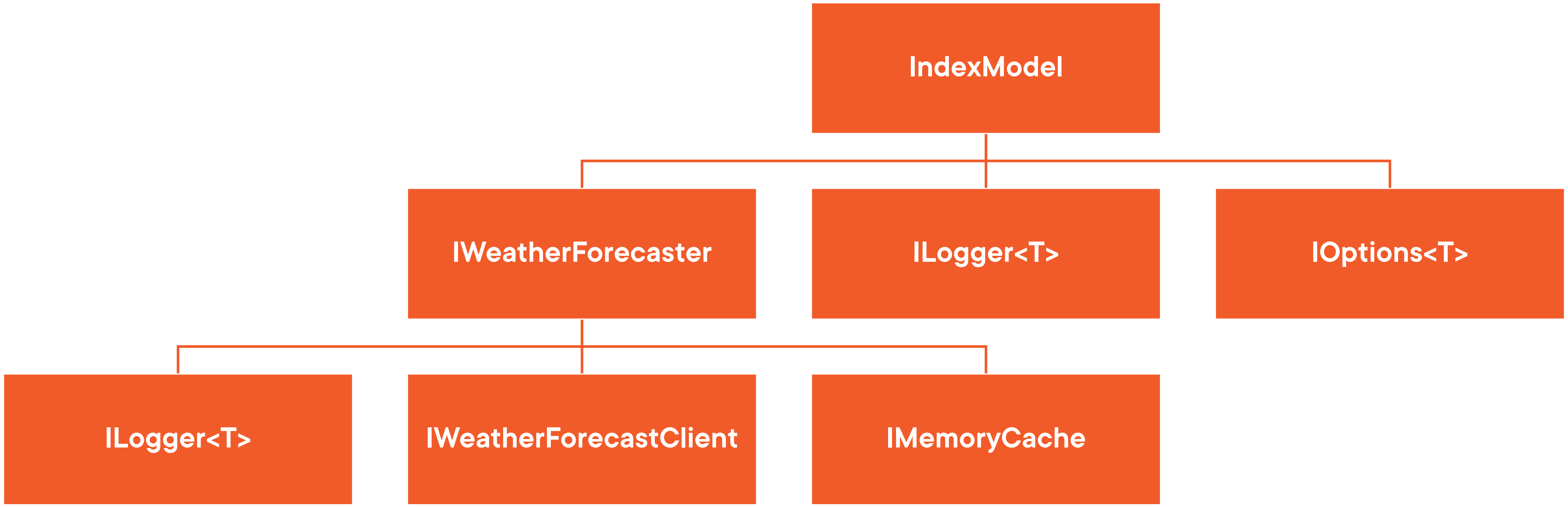
Rinse and repeat

Dependency Graph

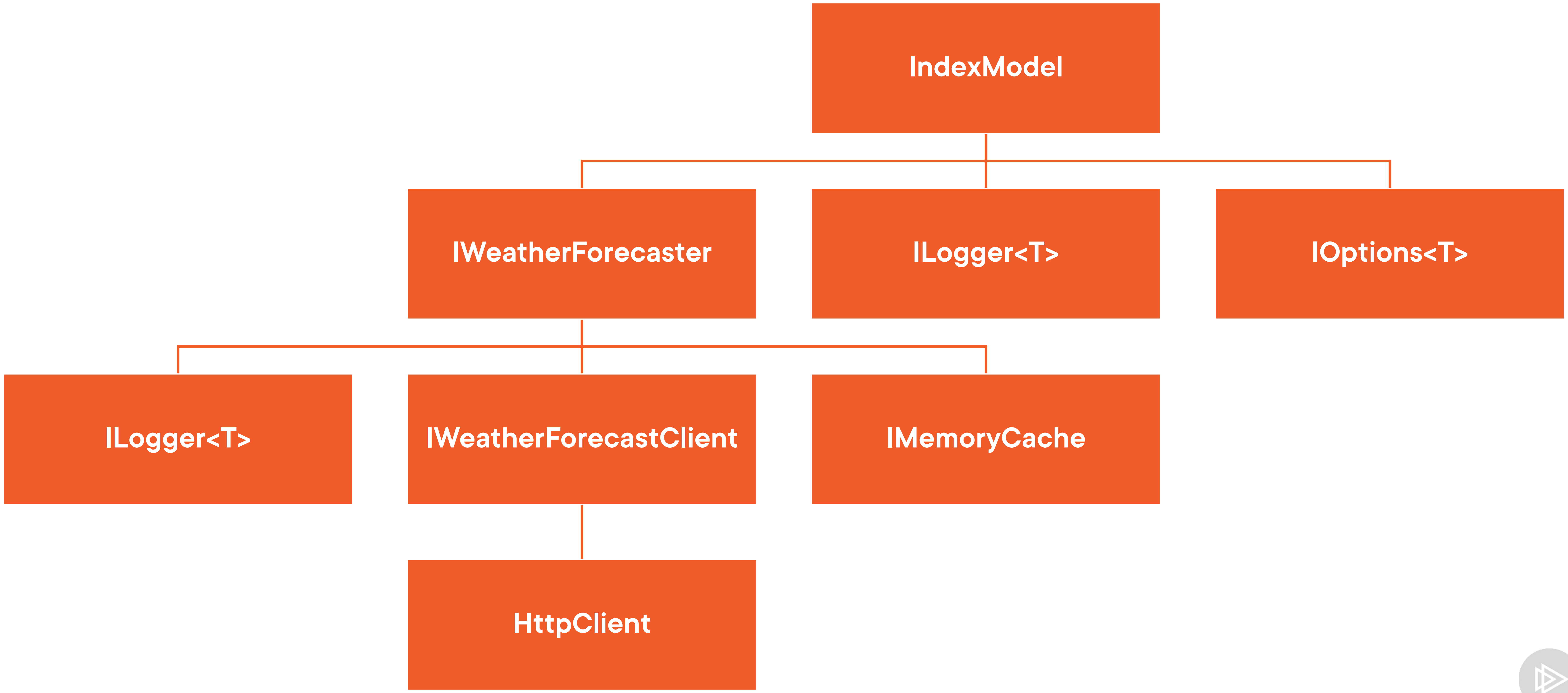
Dependency Graph



Dependency Graph



Dependency Graph



Single Responsibility Principle

Every module, class or function in a computer program should have responsibility over a single part of that program's functionality.

Register all services in the dependency graph

Use constructor injection to accept dependencies

The Microsoft container will manage creation of the object graph

Some usages of the ‘new’ keyword <u>do not</u> identify a dependency.

Plain **O**ld **C**LR **O**bjects

POCO Types

Registering POCO classes is a misuse of the dependency injection pattern

These are not dependencies

They are used mainly as the input or output from methods

They can be created using the ‘new’ keyword

Does the object creation affect testability of the class?

Primitive Types and Strings

Should not be injected or registered in a dependency injection container

Value types (structs) cannot be registered with the container

A common requirement is to provide configuration

Prefer the strongly-typed options pattern

Demo

Best practice for handling configuration

Use built-in ASP.NET Core configuration system and options pattern

Using Configuration and Options in .NET Core and ASP.NET Core Apps

Steve Gordon

app.pluralsight.com/library/courses/dotnet-core-aspnet-core-configuration-options

Service Lifetimes

The service lifetime controls how long resolved objects live.

Registering Lifetimes on the IServiceCollection

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddTransient<IServiceA, ServiceA>();
   services.AddSingleton<IServiceB, ServiceB>();
   services.AddScoped<IServiceC, ServiceC>();
}
```

Service Lifetimes

Transient

Singleton

Scoped

The dependency injection container tracks the instances it creates

Objects are disposed of or released for garbage collection once their lifetime ends

The lifetime affects the creation and reuse of service instances

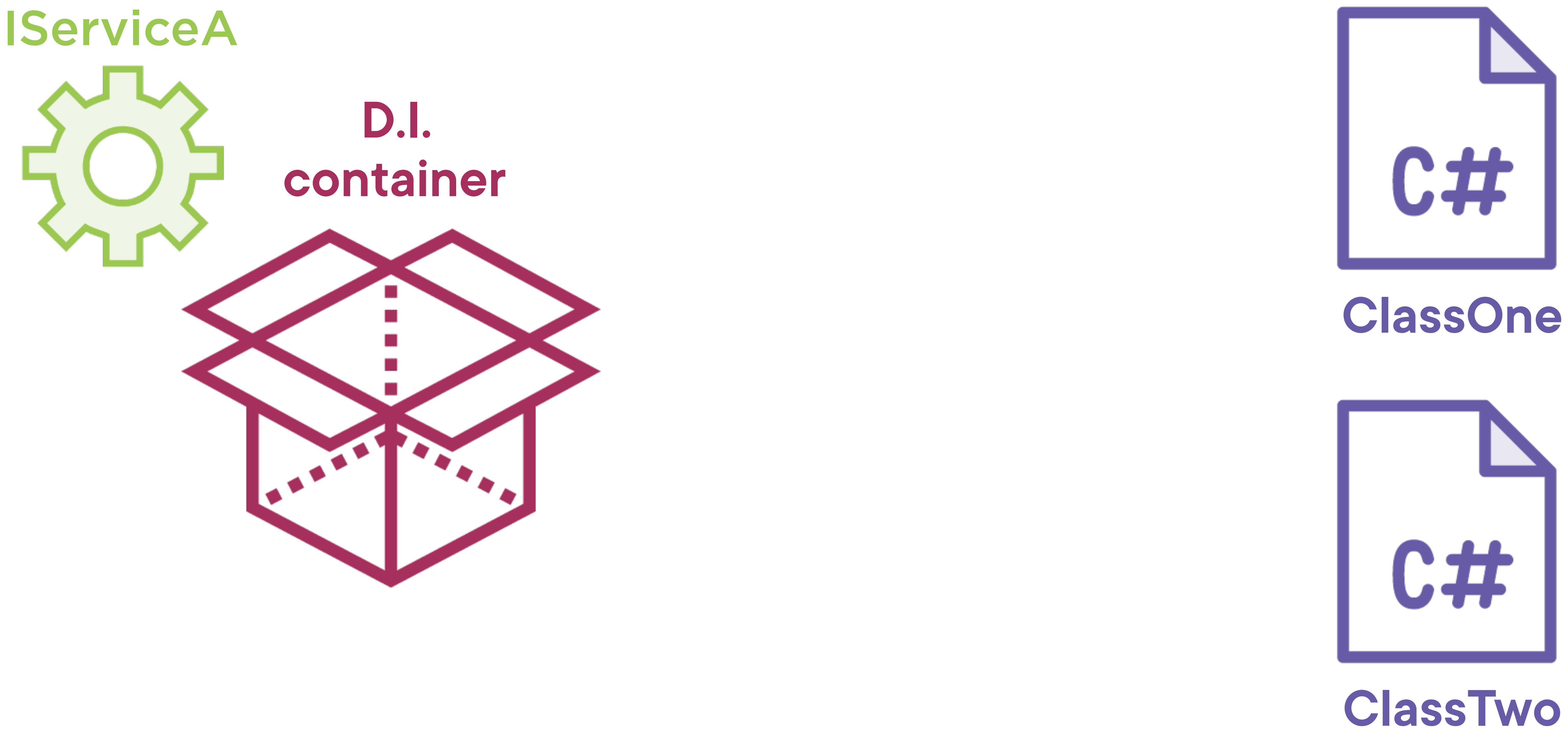
Choose service lifetimes wisely!

Transient Services

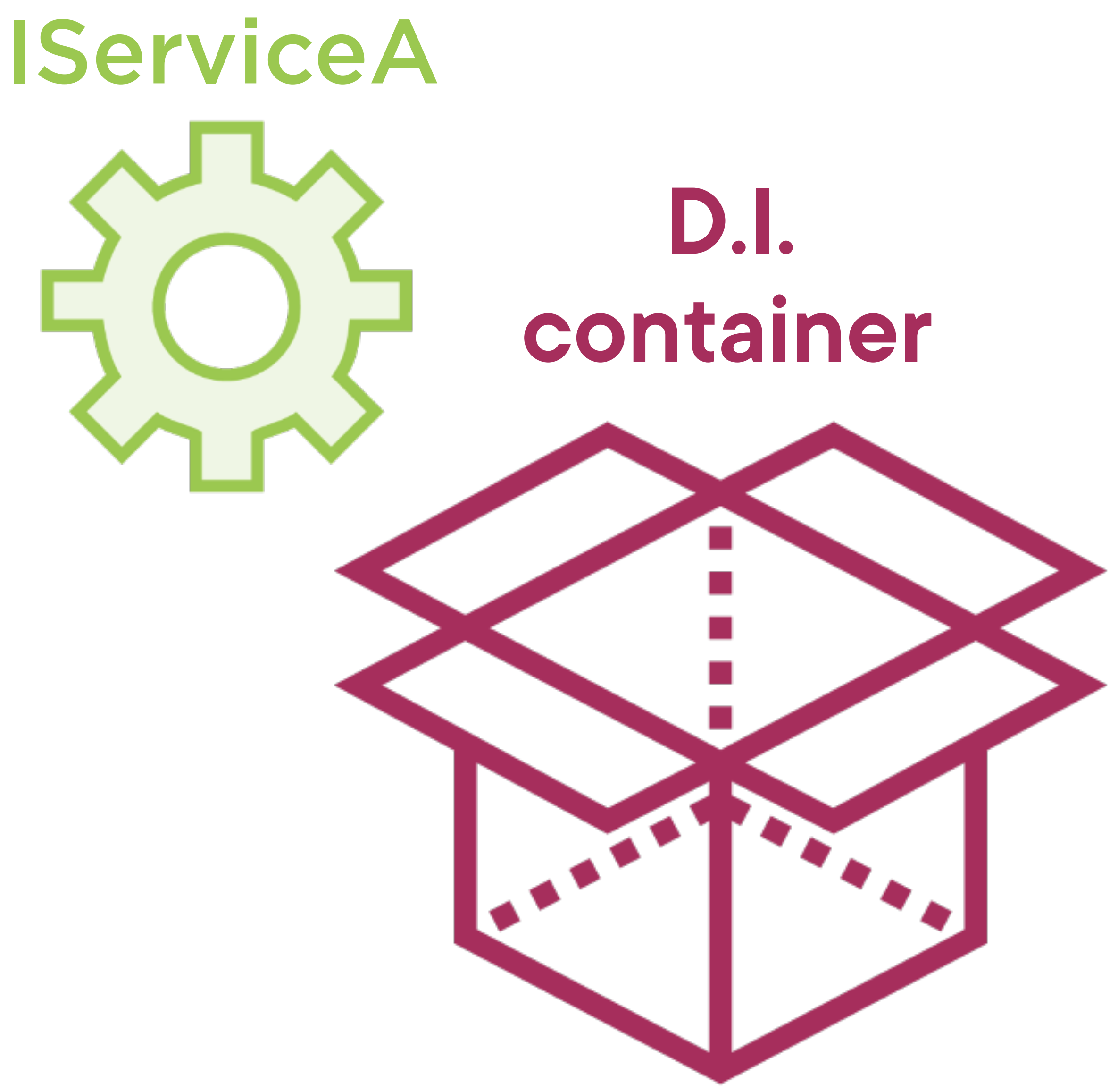
A new instance every time the service is resolved.

Each dependent class receives its own unique instance when the dependency is injected by the container.

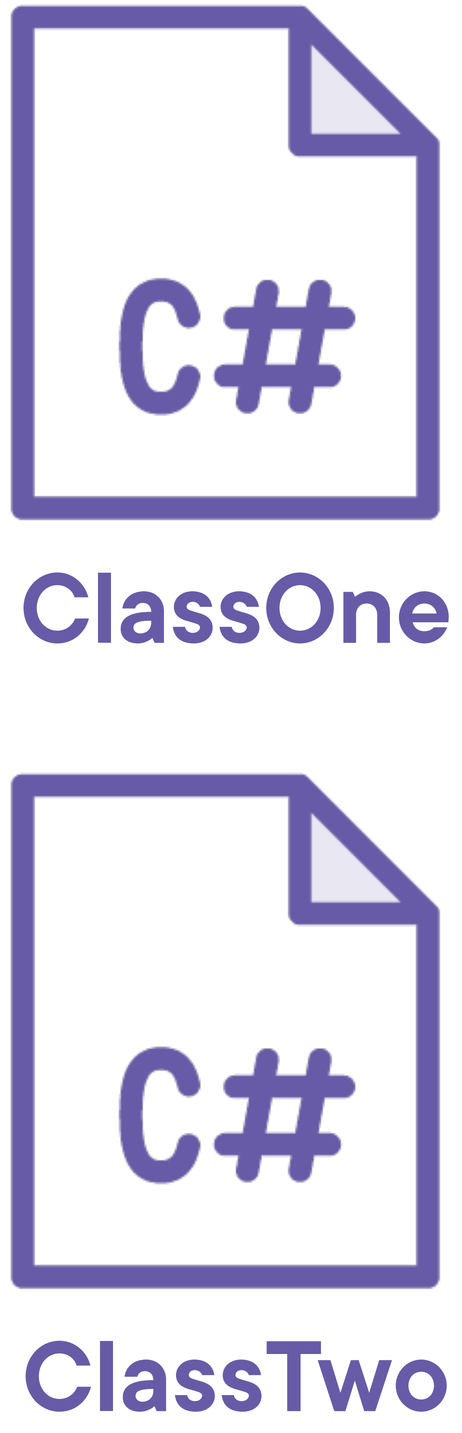
Resolving Transient Services



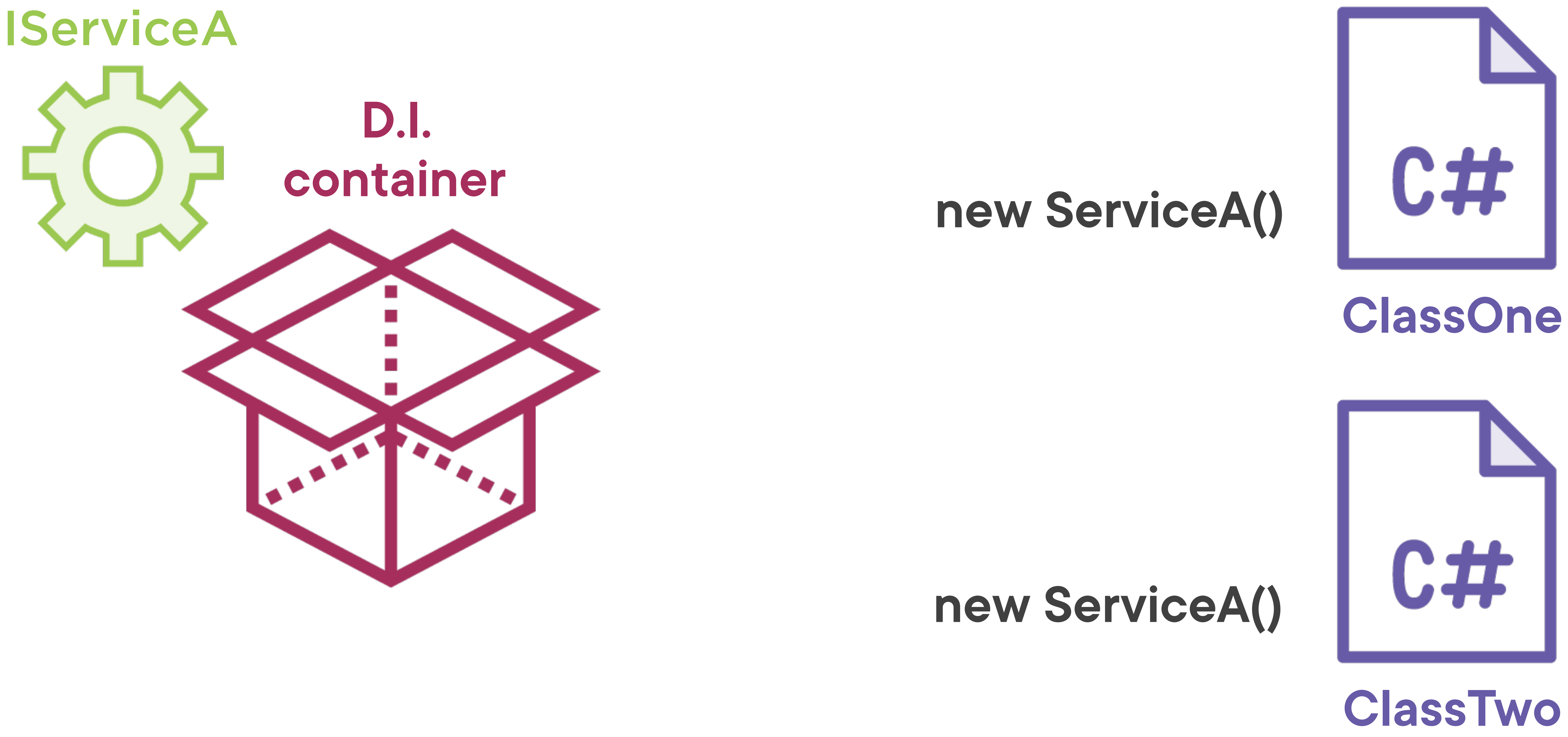
Resolving Transient Services



new ServiceA()



Resolving Transient Services



Use of Transient Services

May contain mutable state

- No requirement to be thread safe

Small performance cost

- Multiple objects are created
- More work for the garbage collector

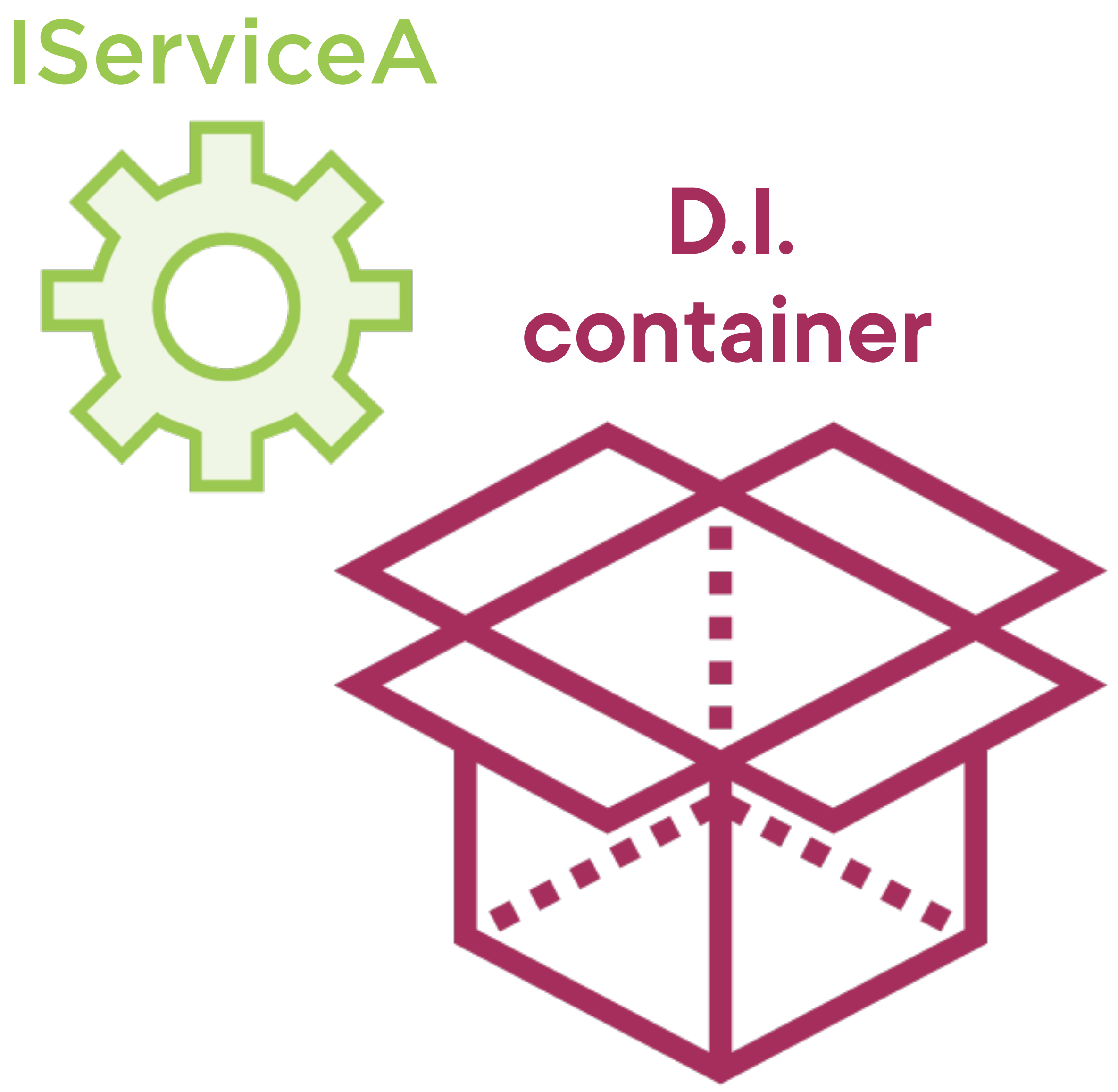
Easiest to reason about

Safest default choice

Singleton Services

One shared instance for the lifetime of the container (application).

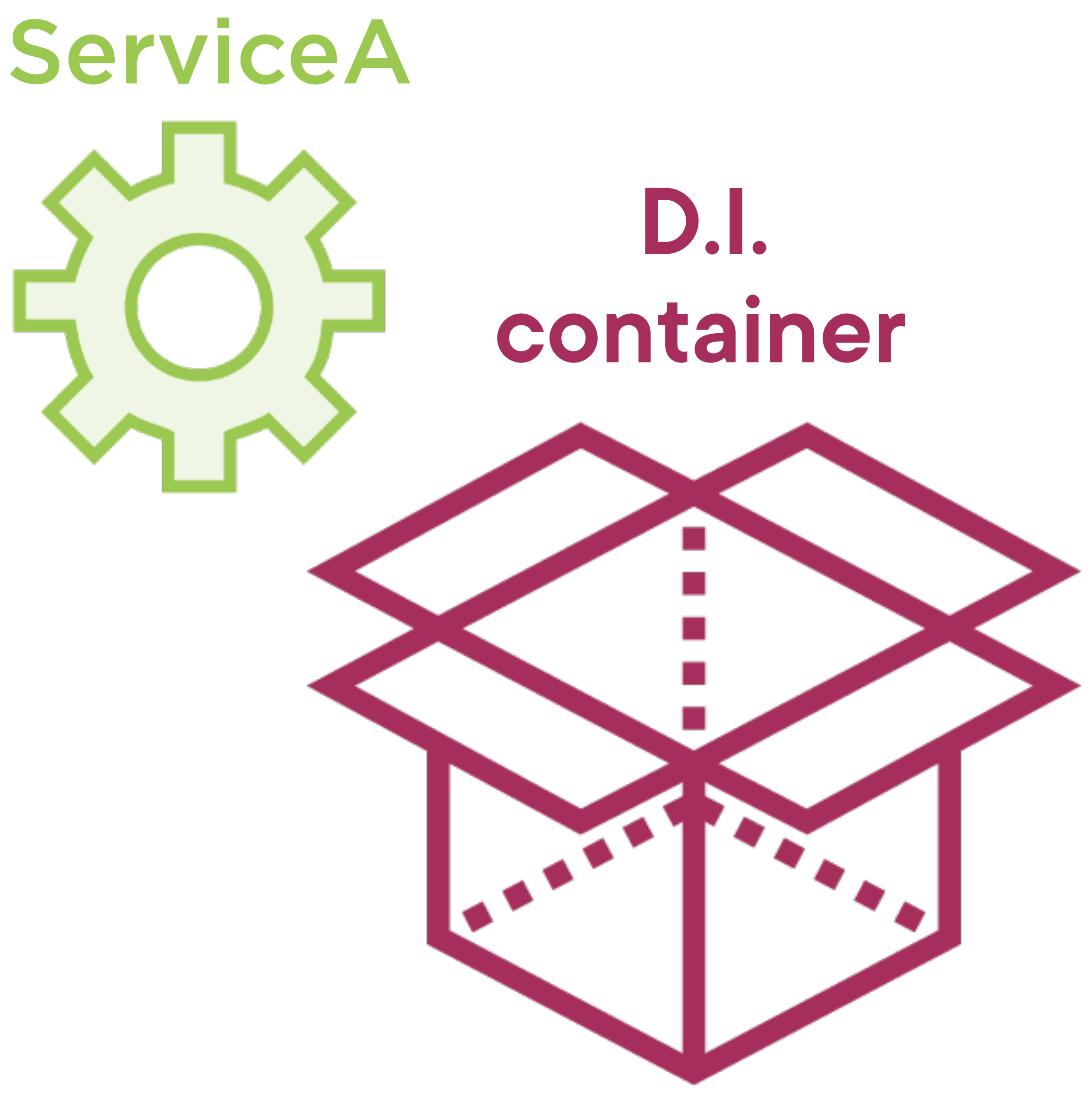
Resolving Singleton Services



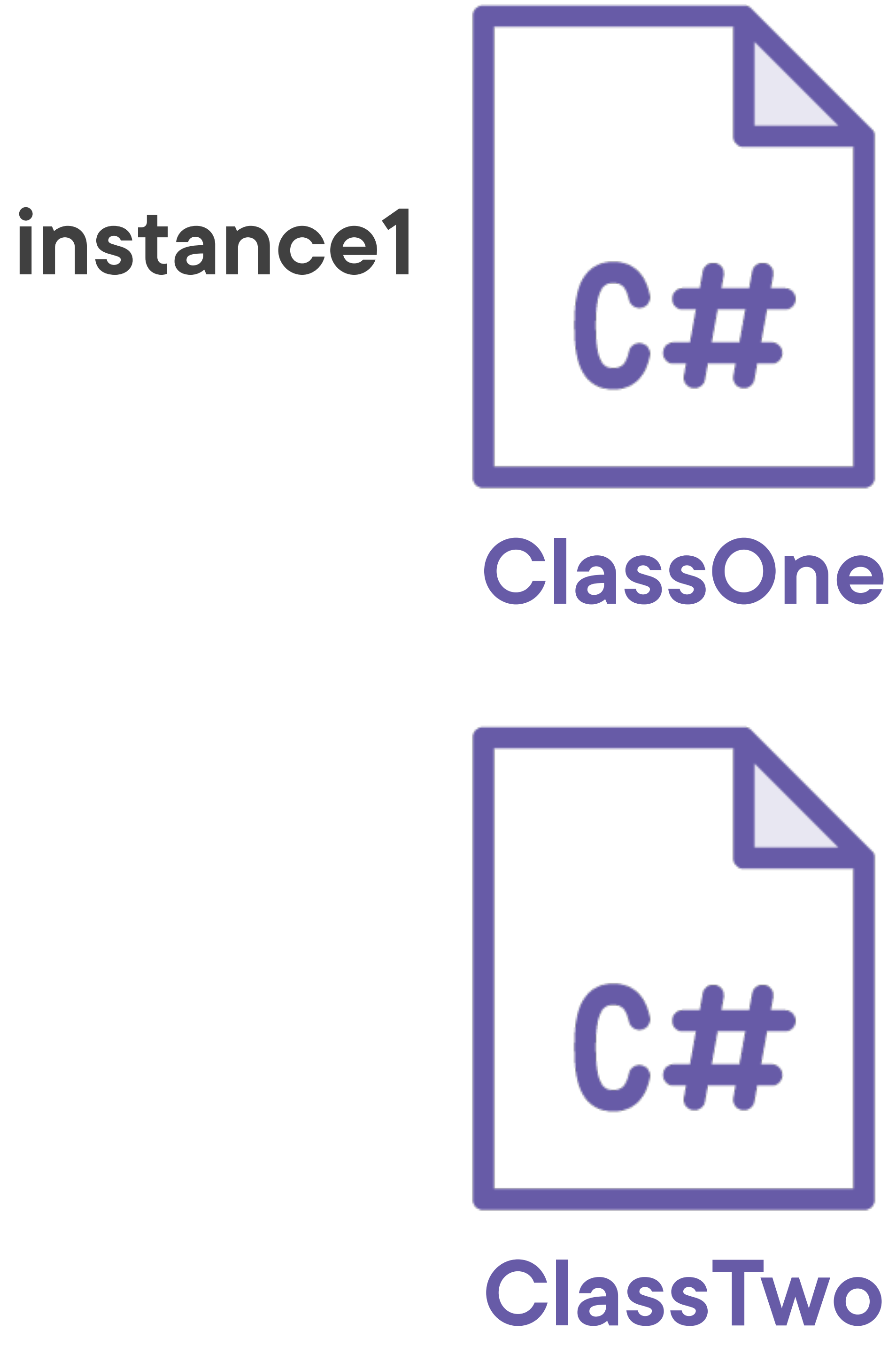
var instance1 =
new ServiceA()

C#
ClassOne
C#
ClassTwo

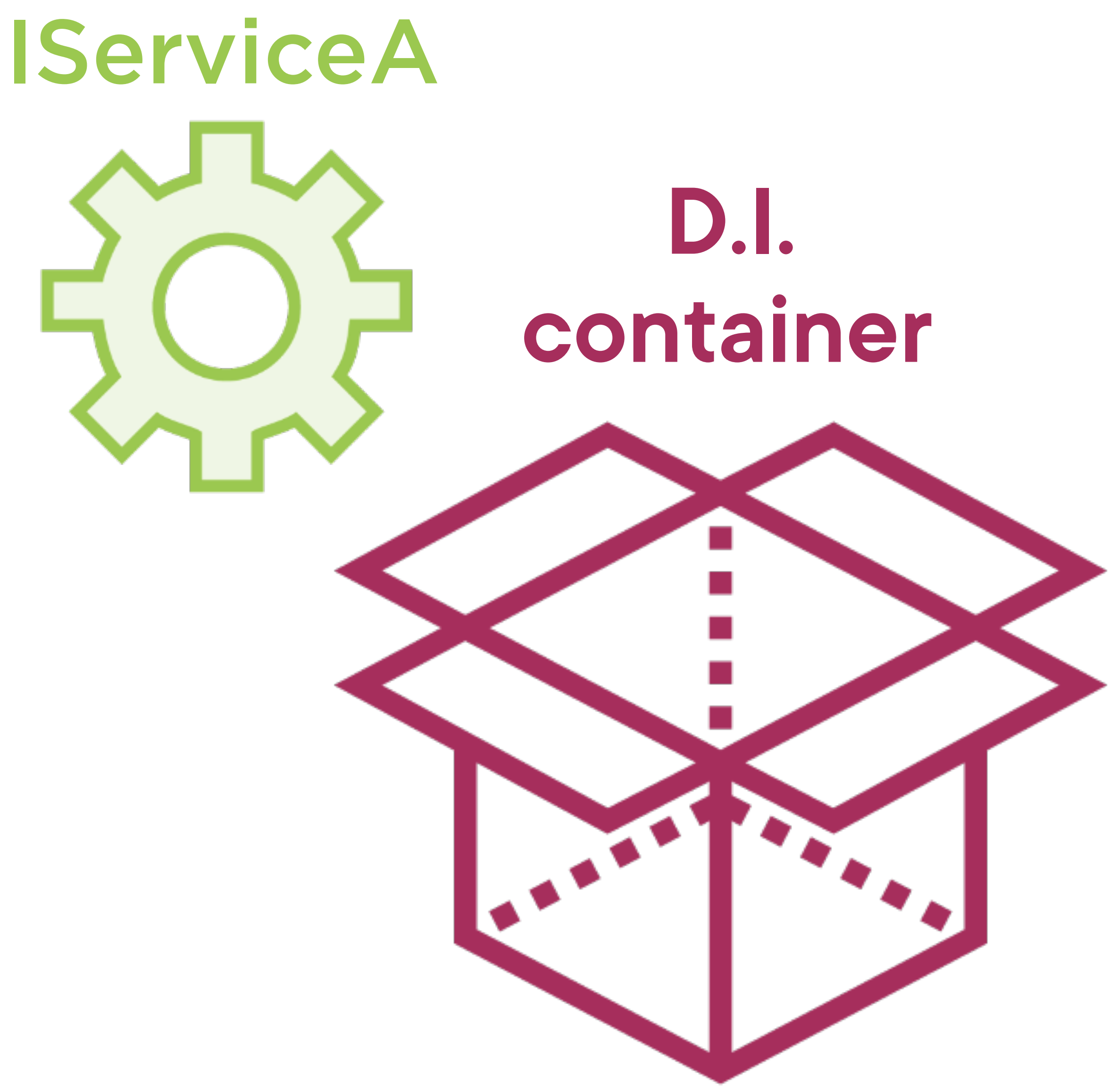
Resolving Singleton Services



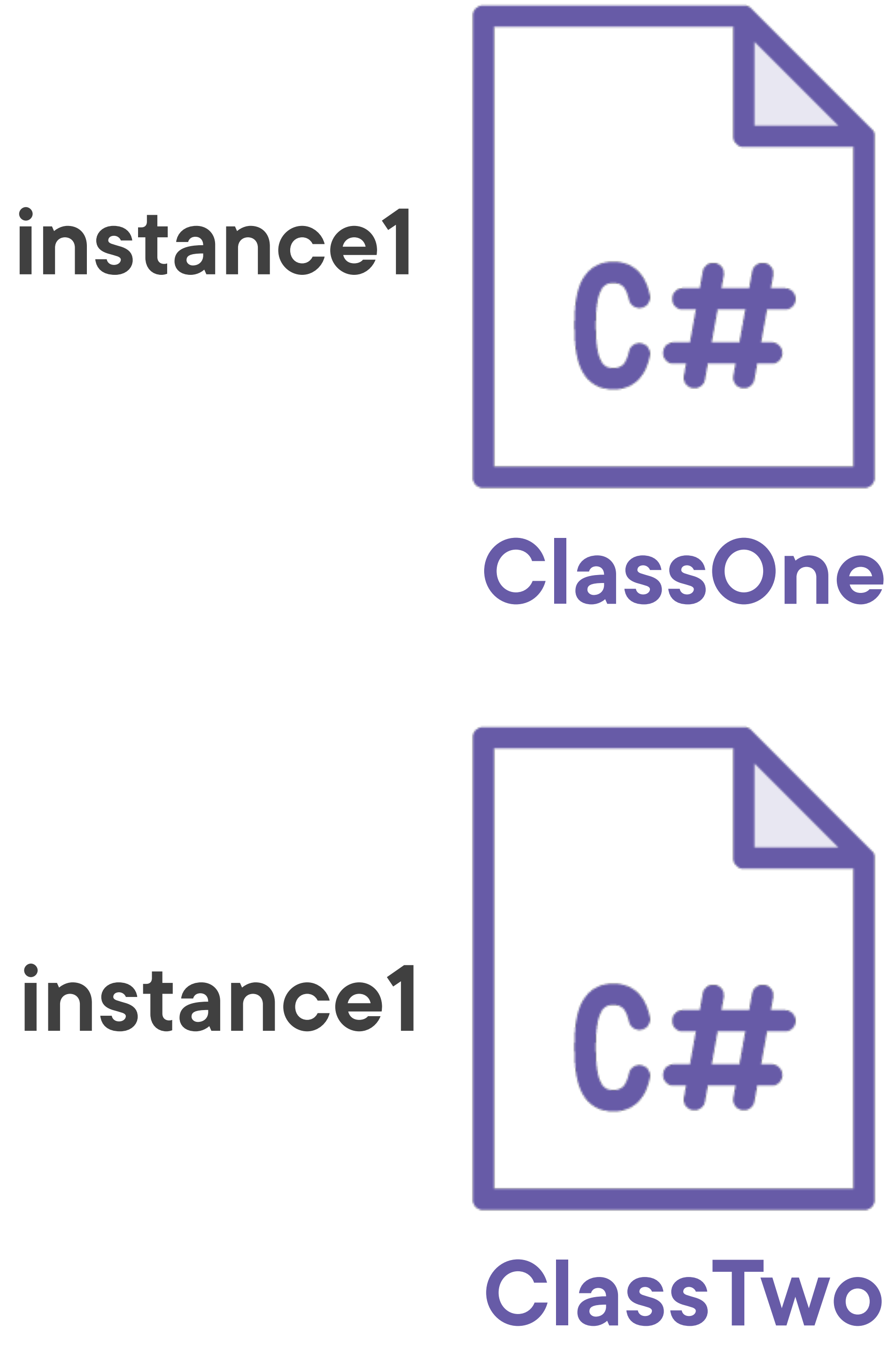
var instance1 =
new ServiceA()



Resolving Singleton Services



var instance1 =
new ServiceA()



Use of Singleton Services

Generally more performant

- Allocates less objects
- Reduces load on GC

Suited to types with expensive or time consuming work at creation

Must be thread-safe

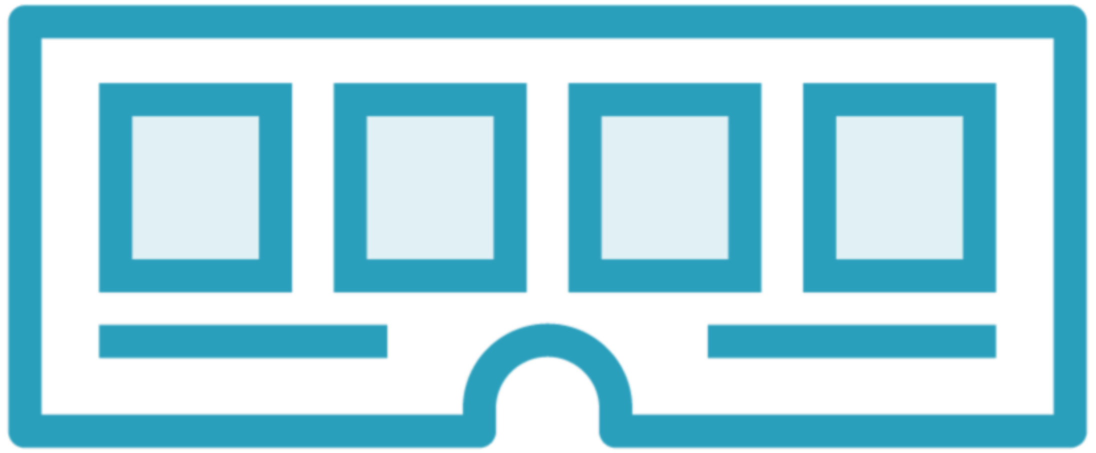
- Avoid mutable state

Suited to:

- Functional stateless services
- Caches

Consider frequency of use vs. memory consumption

Memory Considerations



Possible to create memory leaks using the singleton lifetime

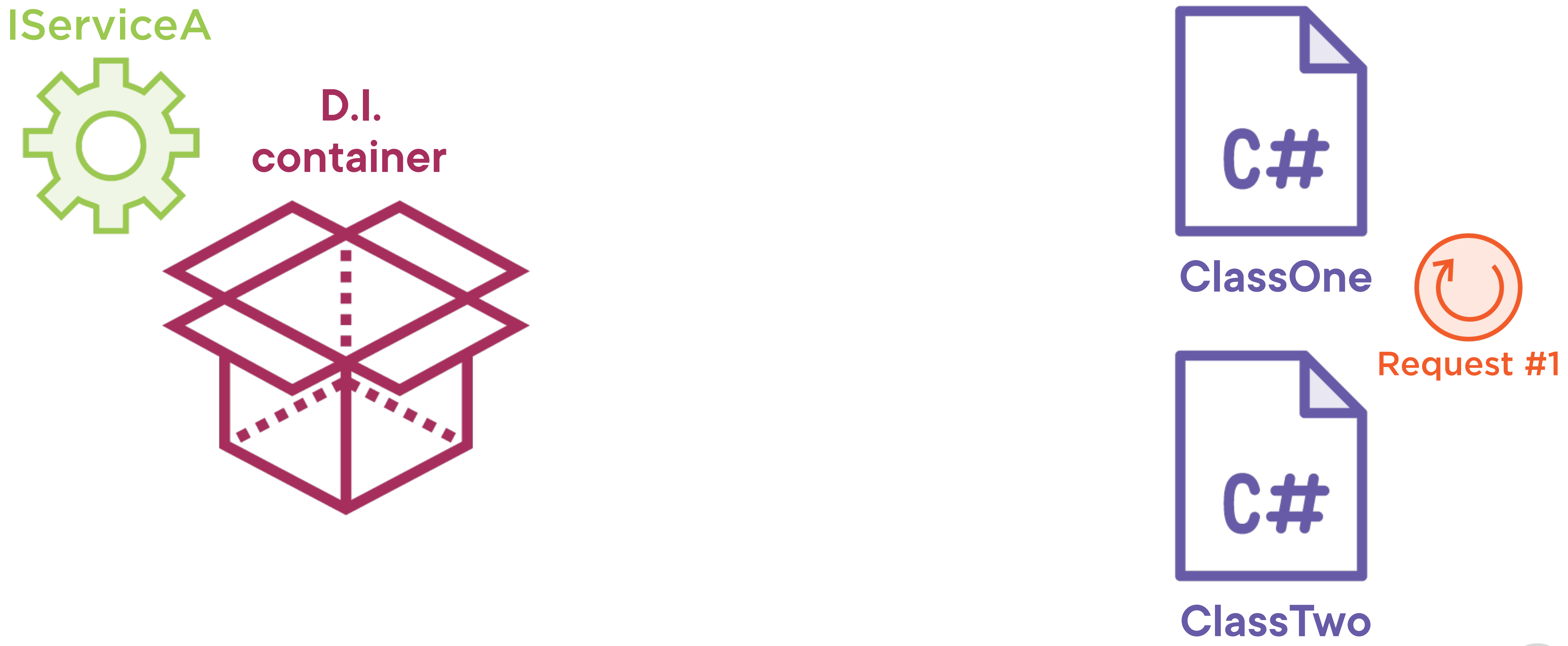
Beware of singleton services where memory usage grows significantly over time

If a service is used very infrequently, the singleton lifetime may not be appropriate

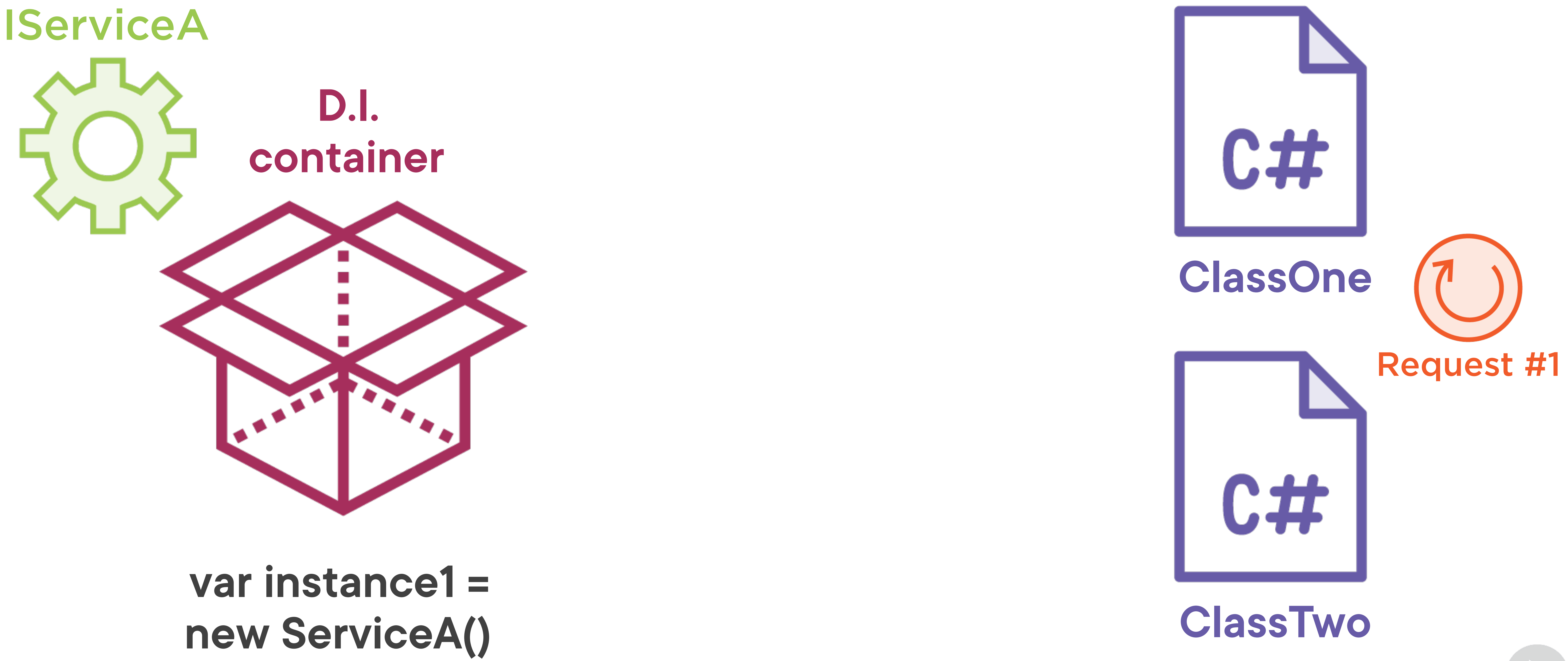
Scoped Services

An instance per scope (request).

Resolving Scoped Services

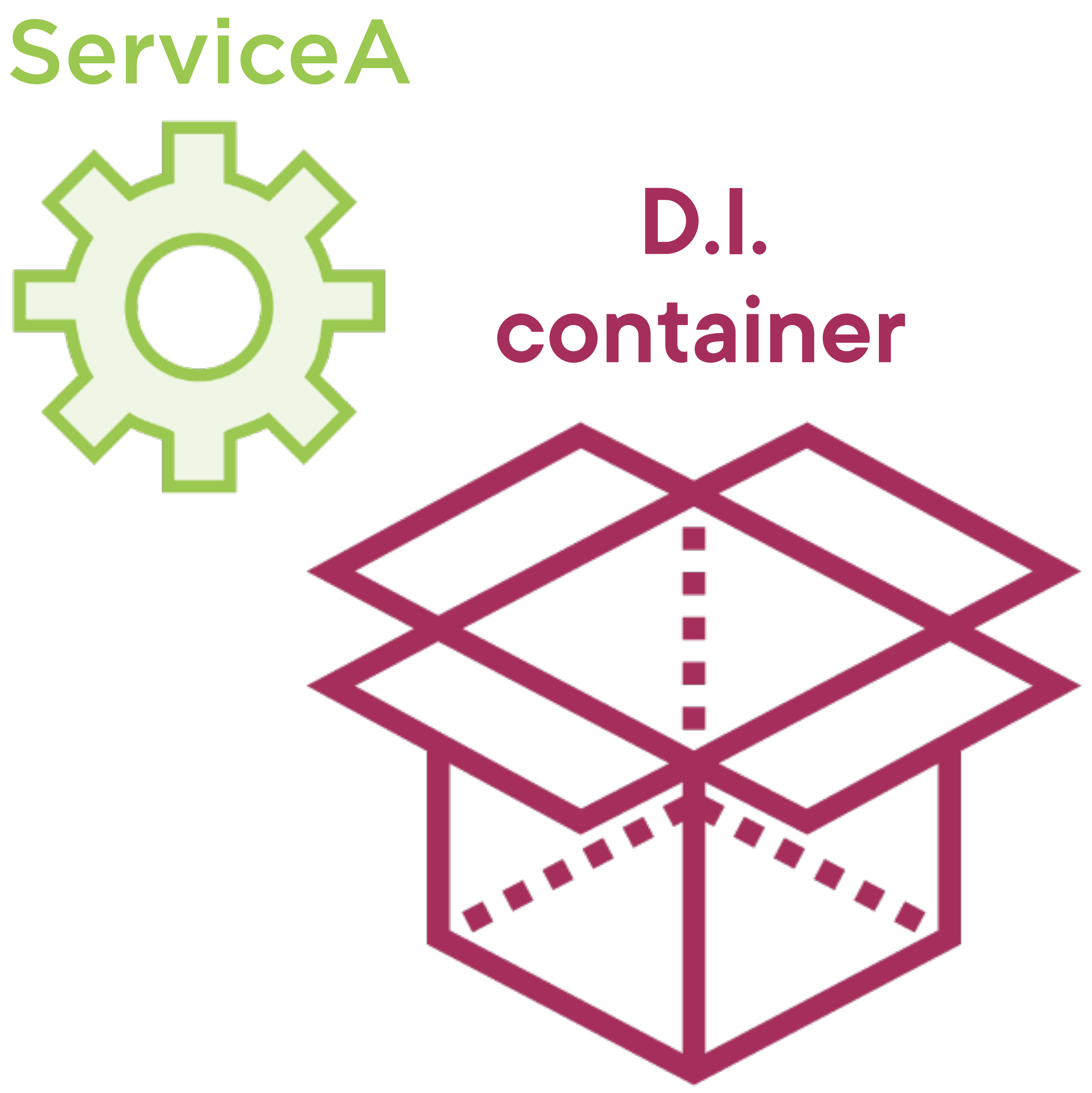


Resolving Scoped Services

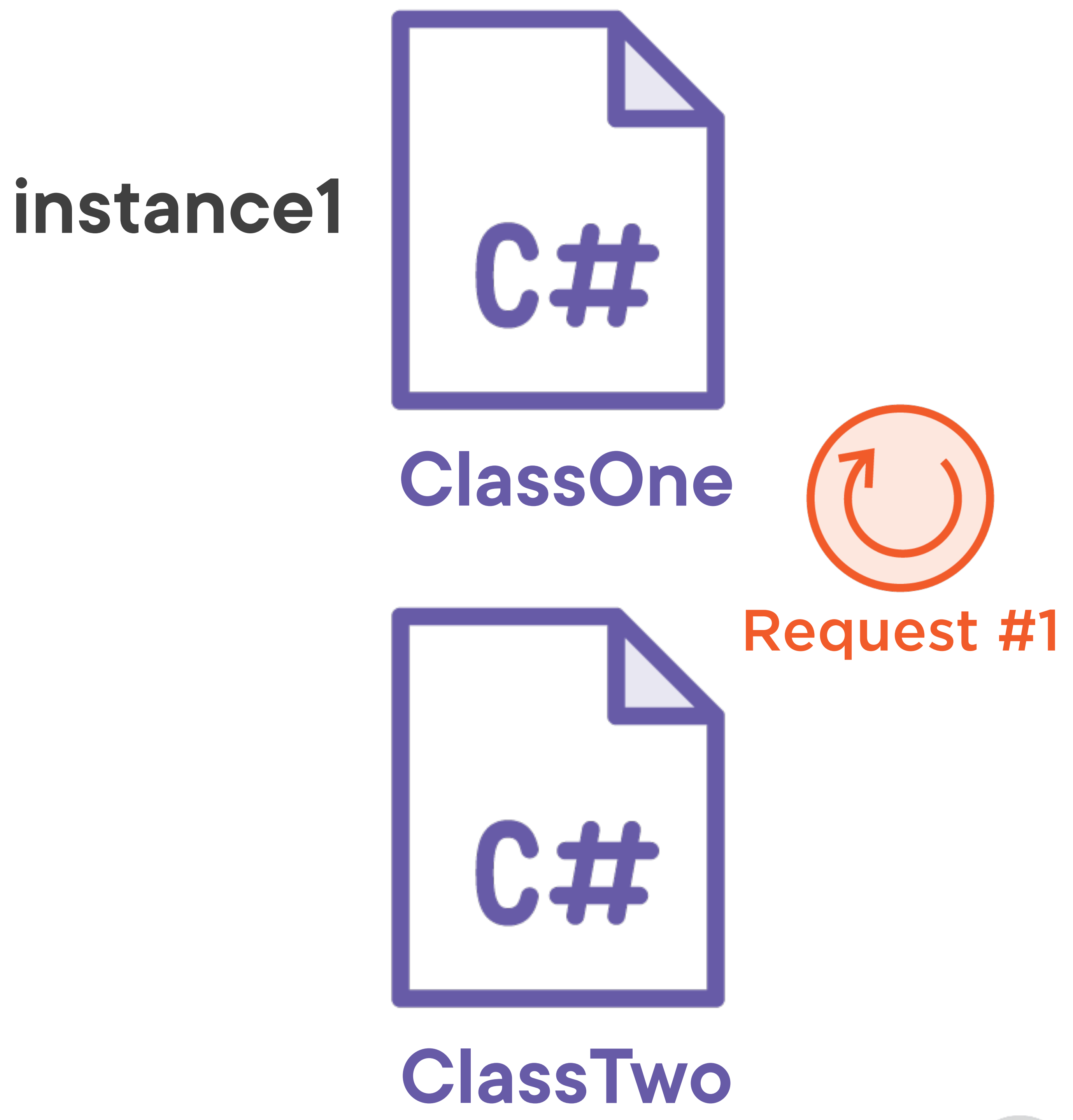


var instance1 = new ServiceA()

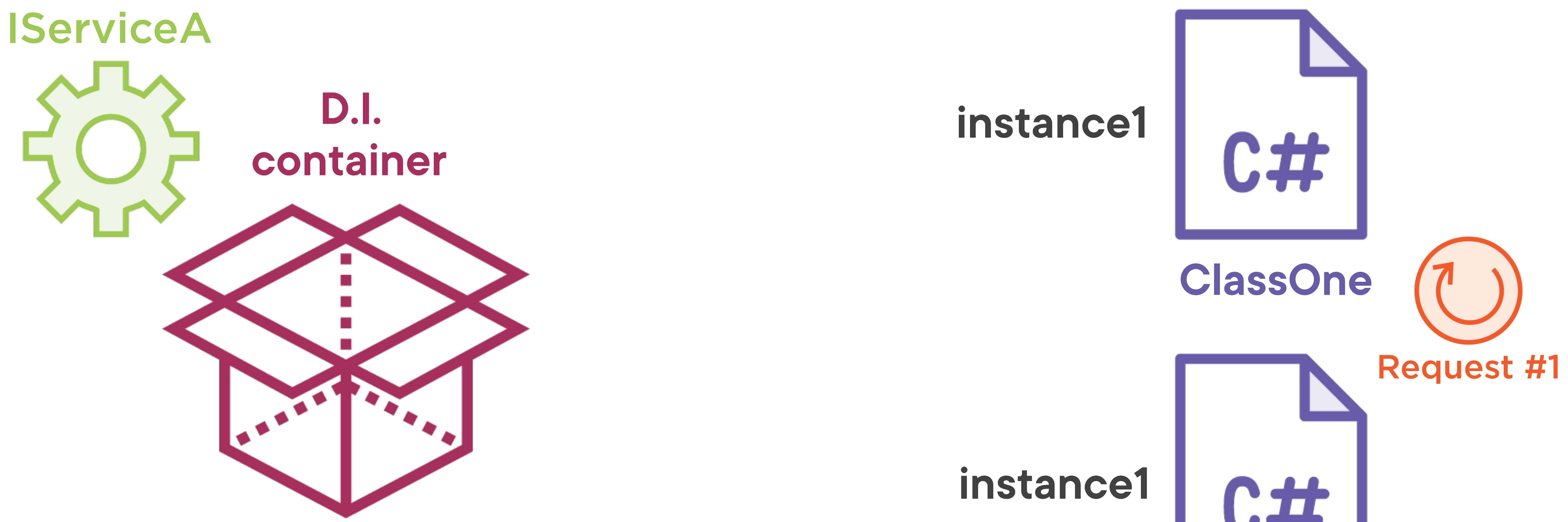
Resolving Scoped Services



var instance1 = new ServiceA()

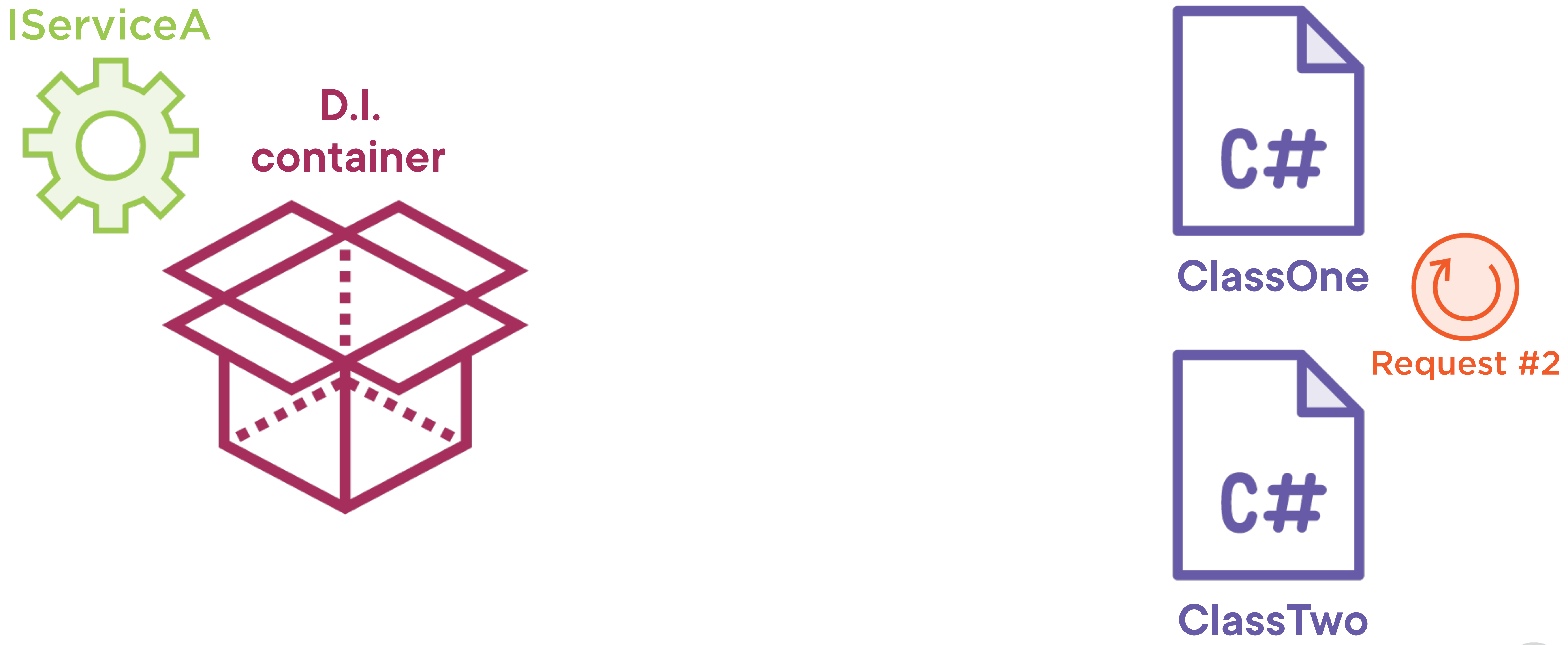


Resolving Scoped Services

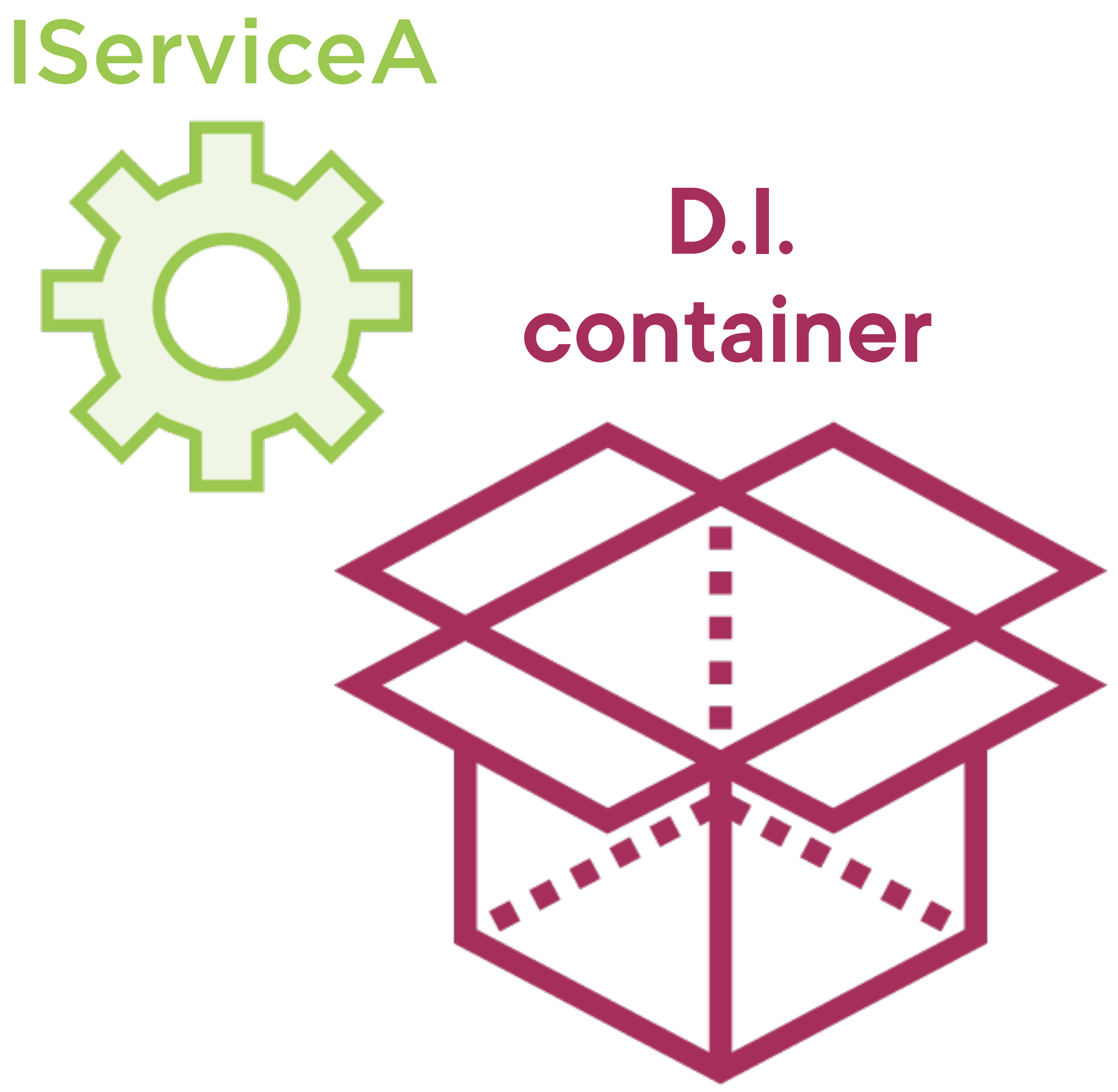


var instance1 =
new ServiceA()

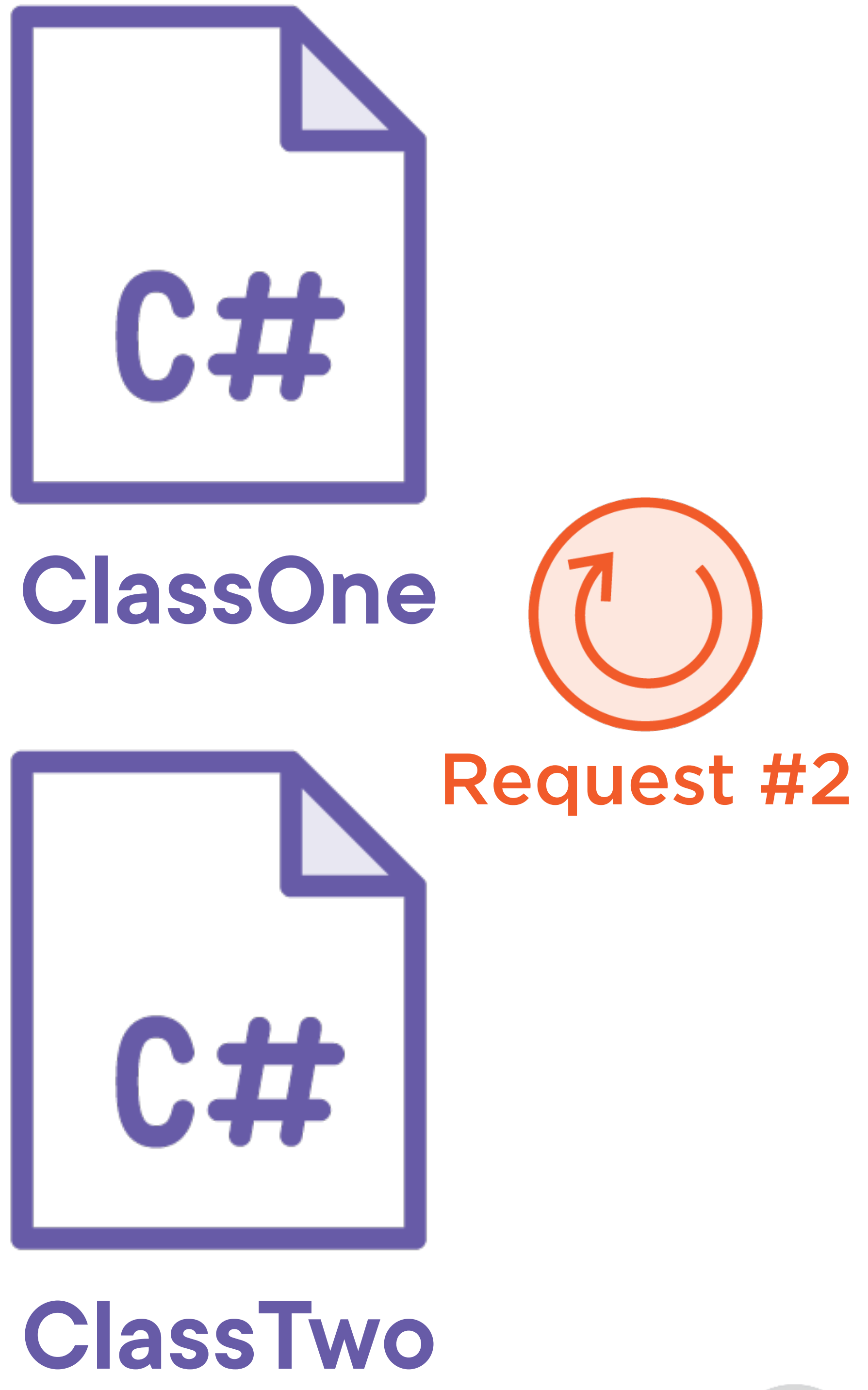
Resolving Scoped Services



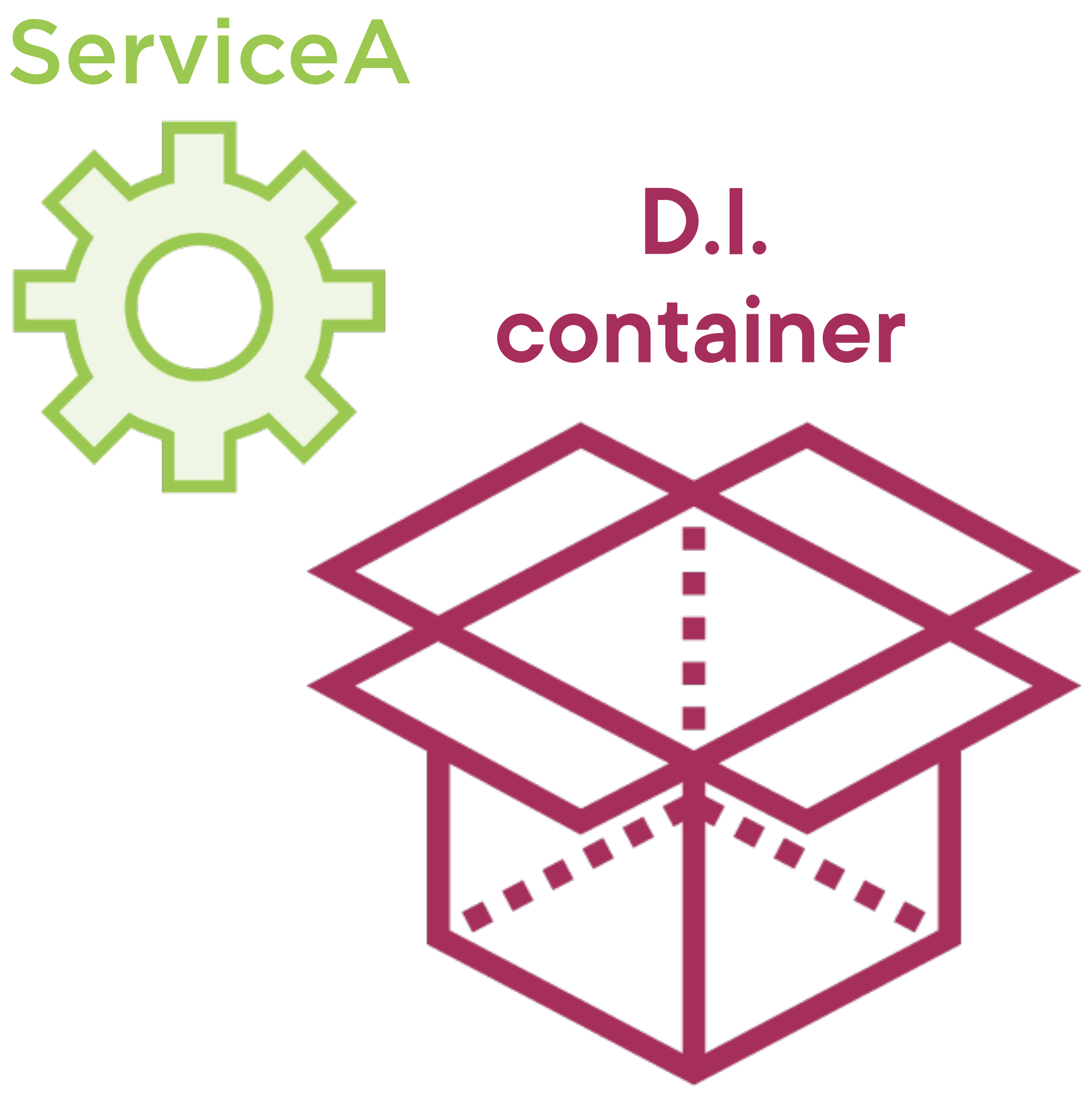
Resolving Scoped Services



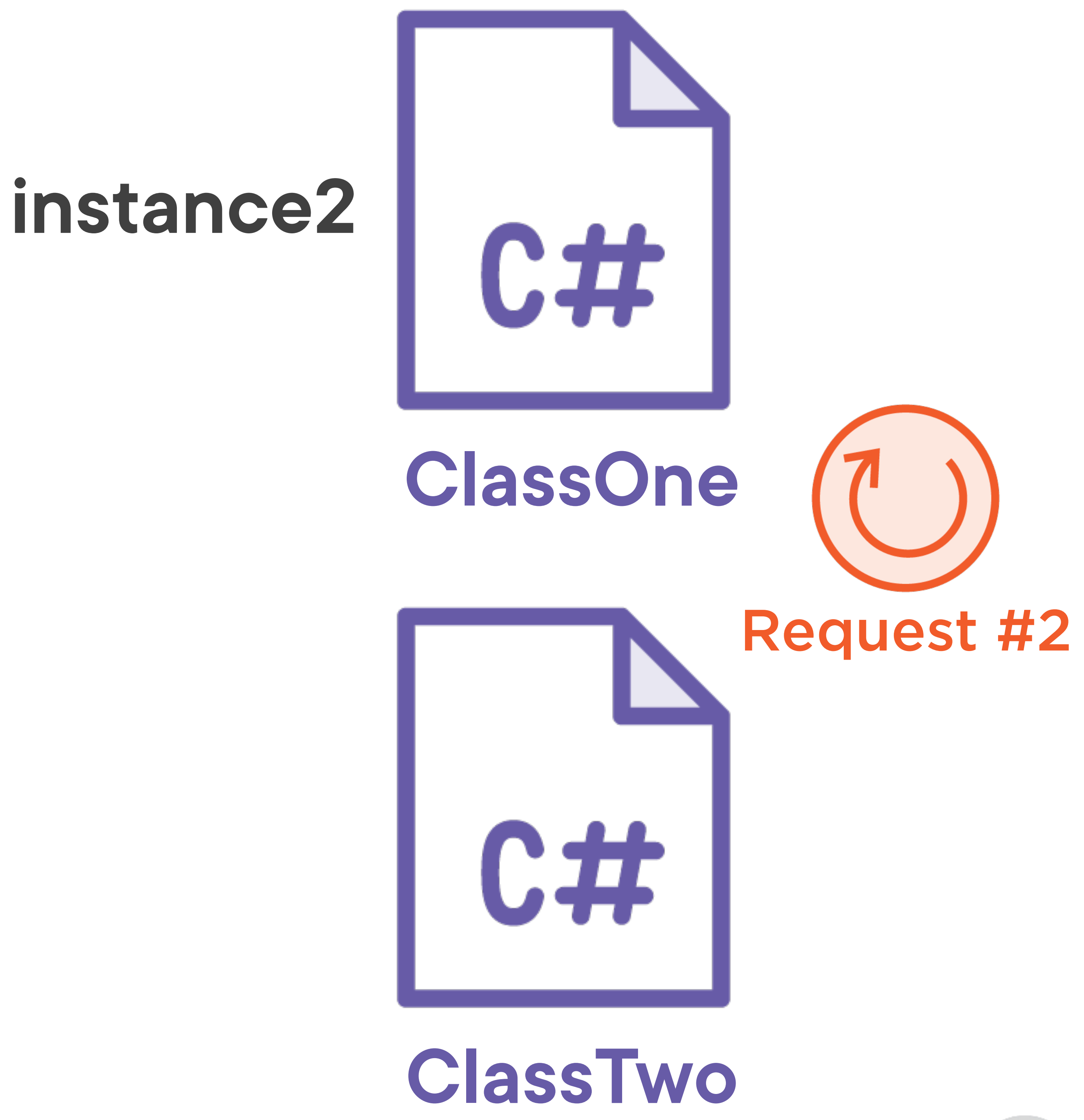
var instance2 = new ServiceA()



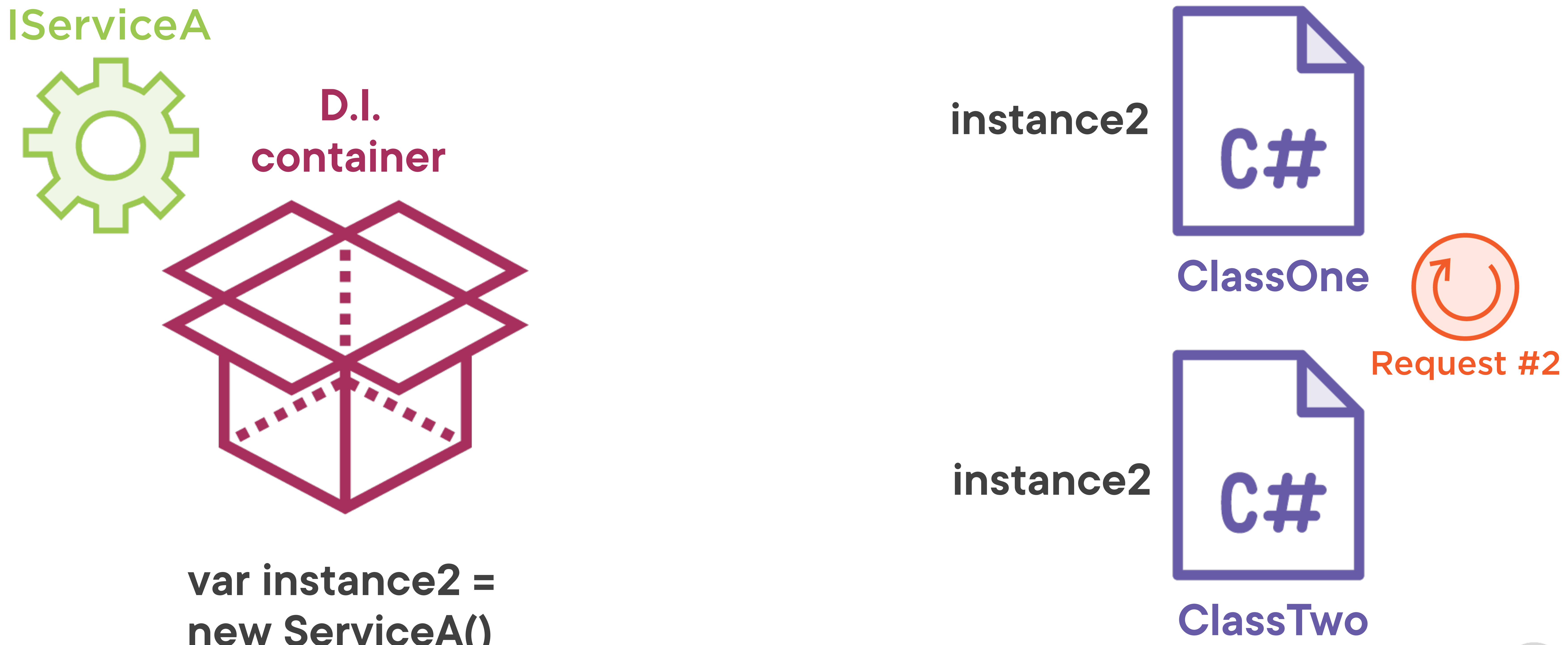
Resolving Scoped Services



var instance2 =
new ServiceA()



Resolving Scoped Services



```
var instance2 =
new ServiceA()
```

Use of Scoped Services

The container creates a new instance per request

- Not required to be thread-safe

Components used in the request lifecycle receive the same dependency instance

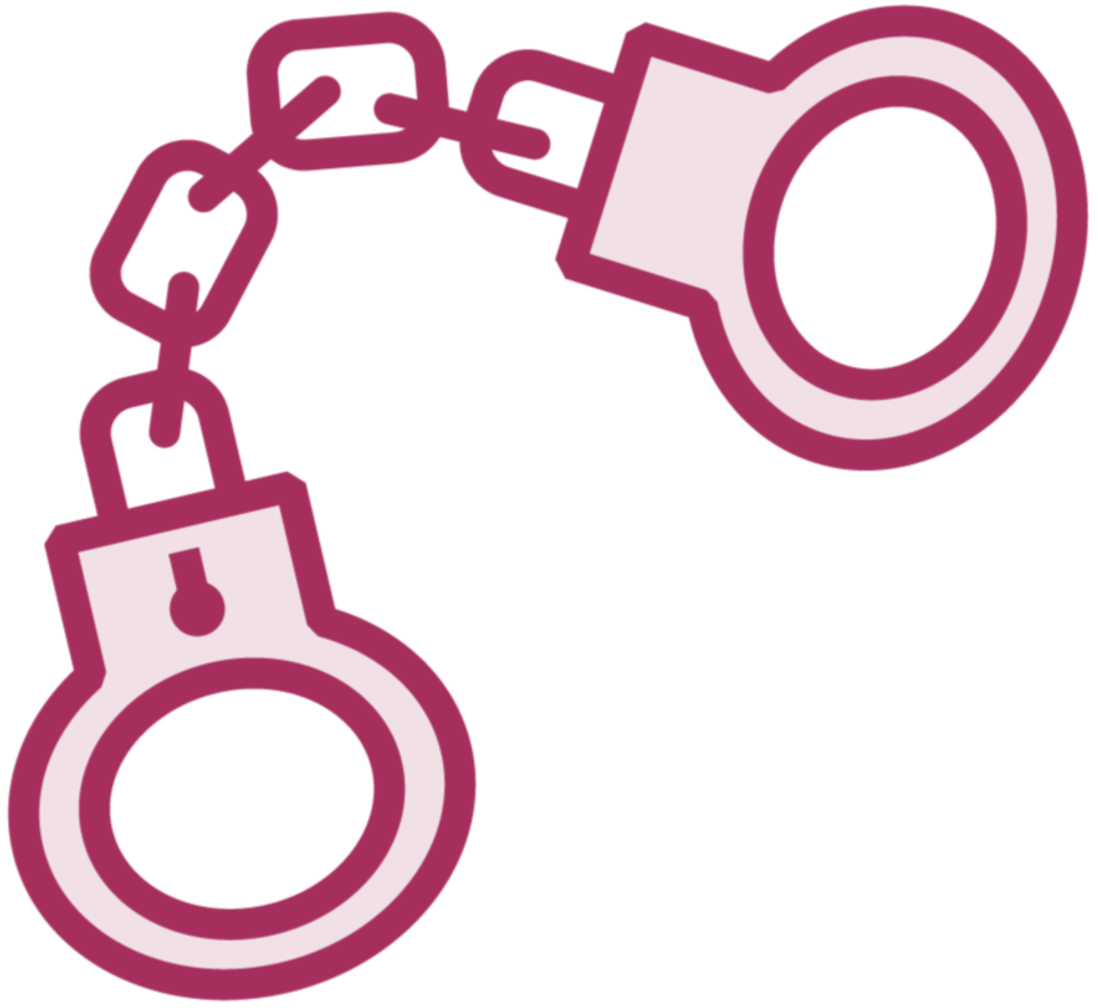
Useful if a service may be required by multiple consumers per request

- An example of a scoped service is the Entity Framework DbContext
- DbContext change tracking works across a single request

Should not be captured by singleton services

Avoiding Captive Dependencies

Captive Dependencies



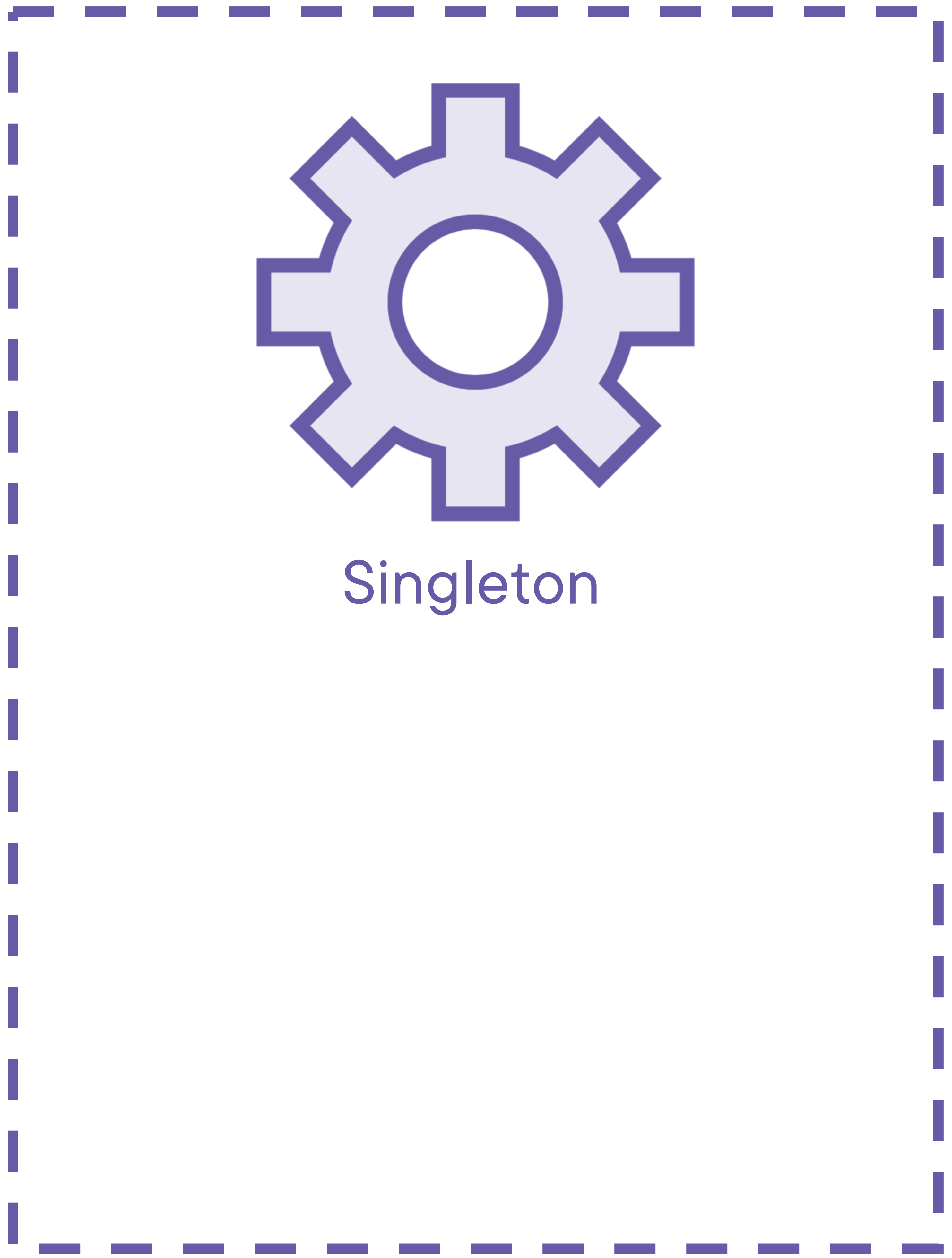
Ensure that the service lifetime is appropriate

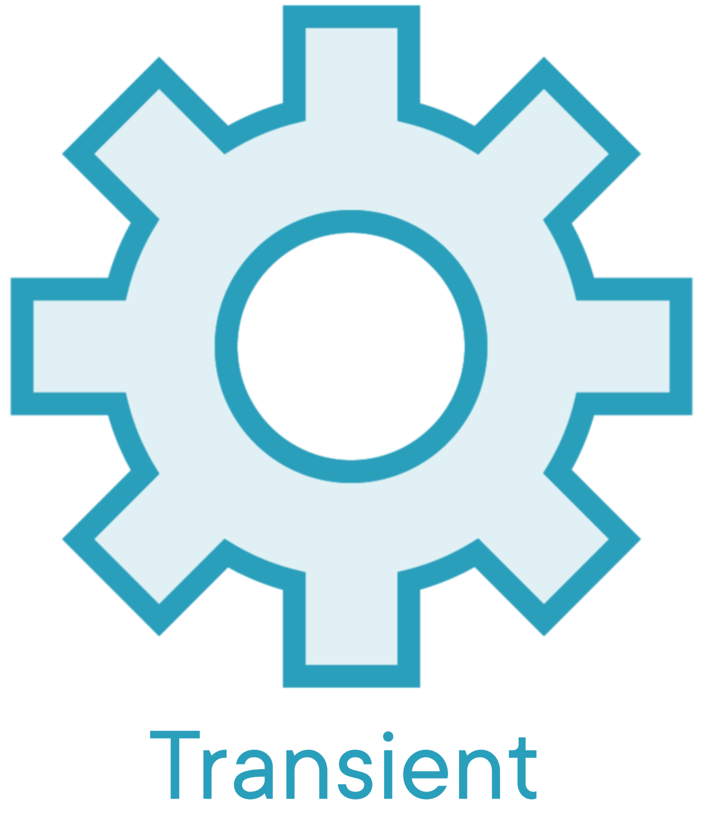
- Consider the lifetime of dependencies

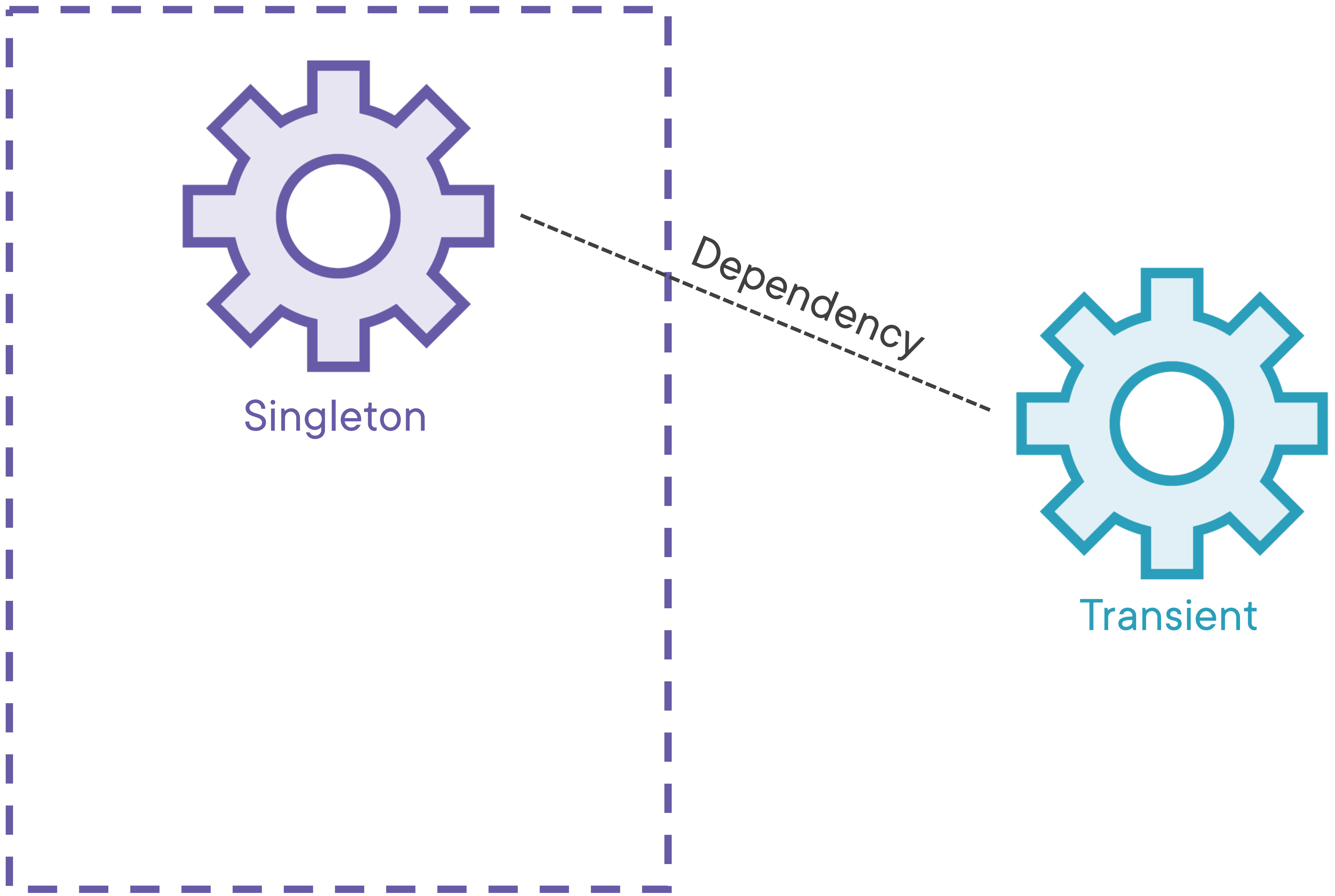
Captive dependencies

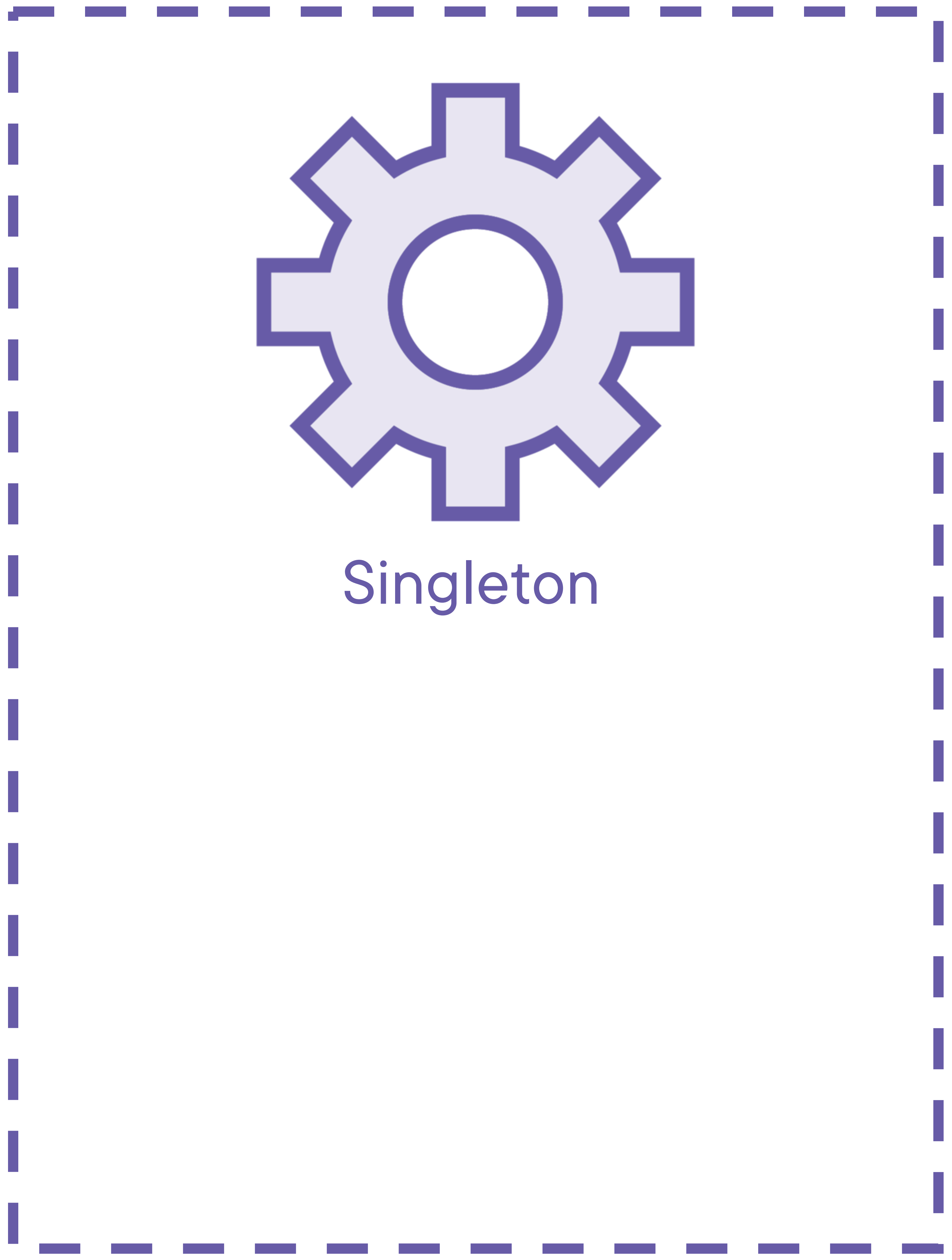
- May live for longer than intended

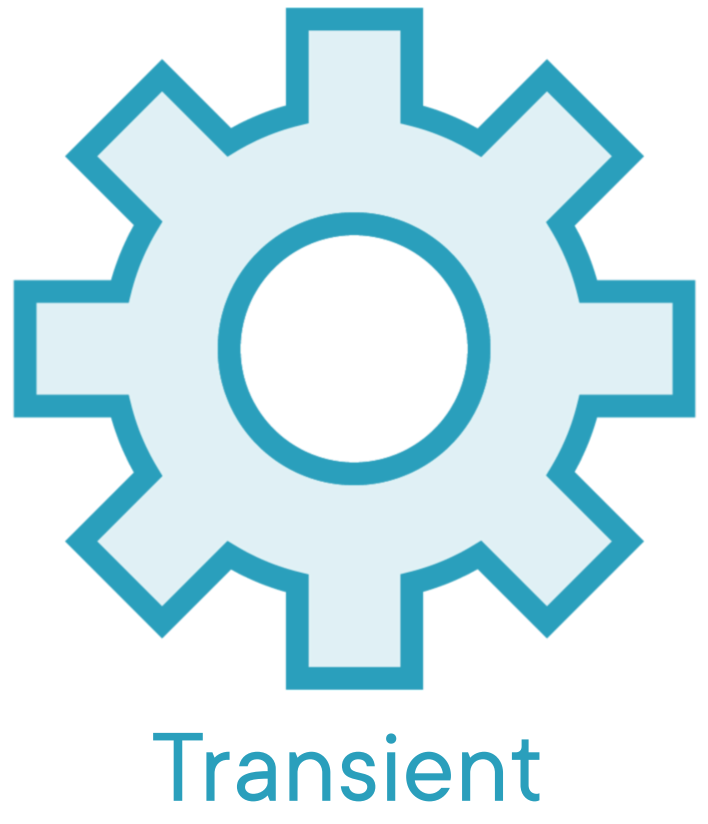
A service should not depend on a service with a lifetime shorter than its own.

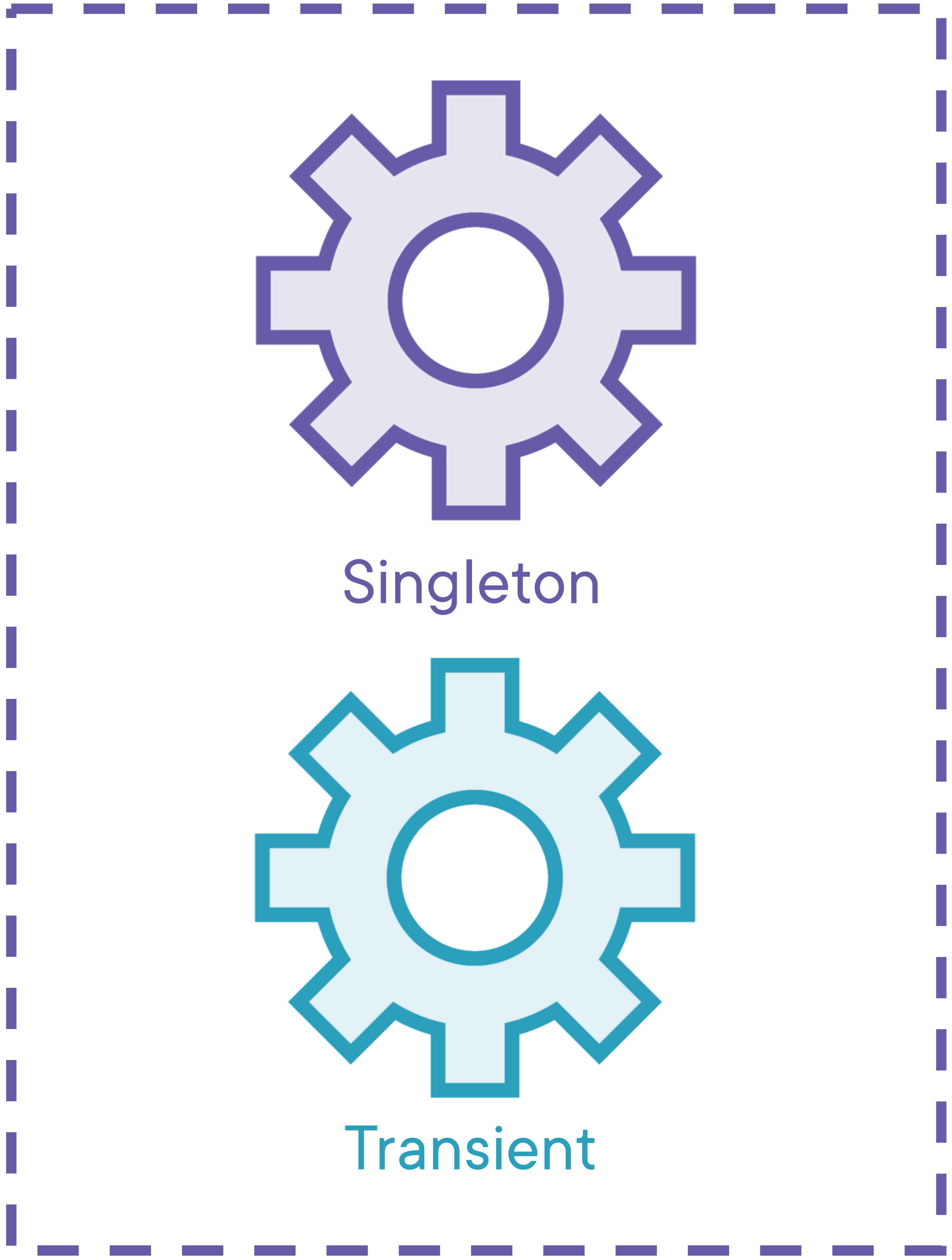

Singleton


Transient


Singleton
Dependency
Transient


Singleton


Transient


Singleton
Transient

Side-effects of Captured Dependencies

Accidental sharing of non-thread-safe services between threads

Objects living longer than their expected lifetime

Safe Dependencies

	Transient	Scoped	Singleton
Transient	✓	✓	✓
Scoped	✗	✓	✓
Singleton	✗	✗	✓

Scope Validation

Scope Validation

Enabled by default in development

Validates container scopes

Validation occurs at startup when the build method is invoked

Any captured scoped services cause a runtime InvalidOperationException

Scope Validation in Production

Disabled by default in production

Lack of a runtime exception does not mean everything is okay

Disposal of Services

```
public interface IDisposable
{
    void Dispose();
}
```

IDisposable Interface

Provides a mechanism for releasing unmanaged resources.

Dispose Pattern

IDisposable Best Practices for C# Developers

Elton Stoneman

app.pluralsight.com/library/courses/c-sharp-developers-idisposable-best-practices

Disposable Types

A using block or using statement is used to signal release of a disposable type

- The compiler generates code which calls Dispose( ) as soon as the consuming code no longer needs it

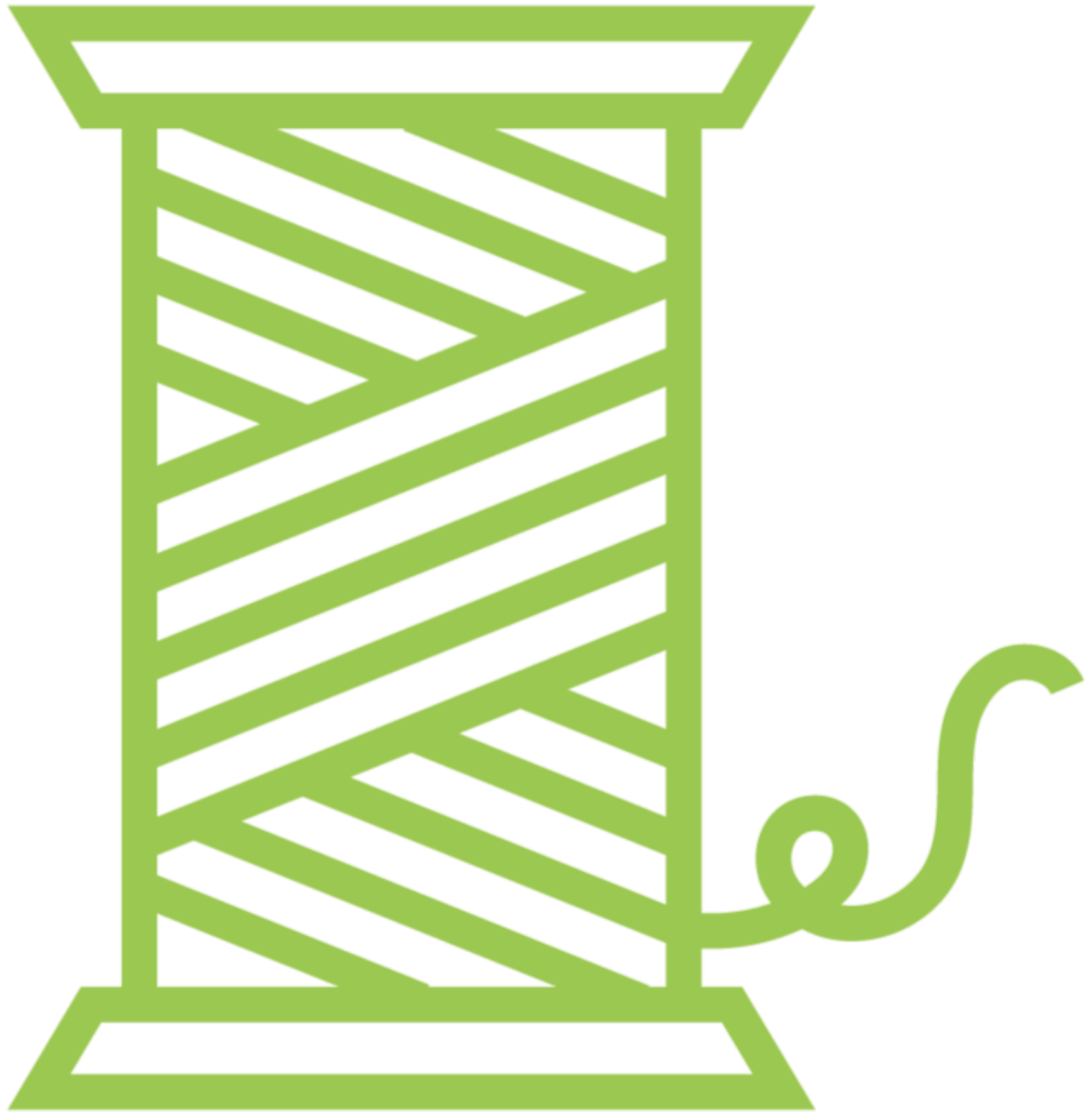
The D.I container supports IDisposable types

- Calls Dispose on instances at the end of their lifetime
- Automatic for types created by (owned by) the container

User created instances are not disposed

- Their lifetime is managed externally

IAsyncDisposable

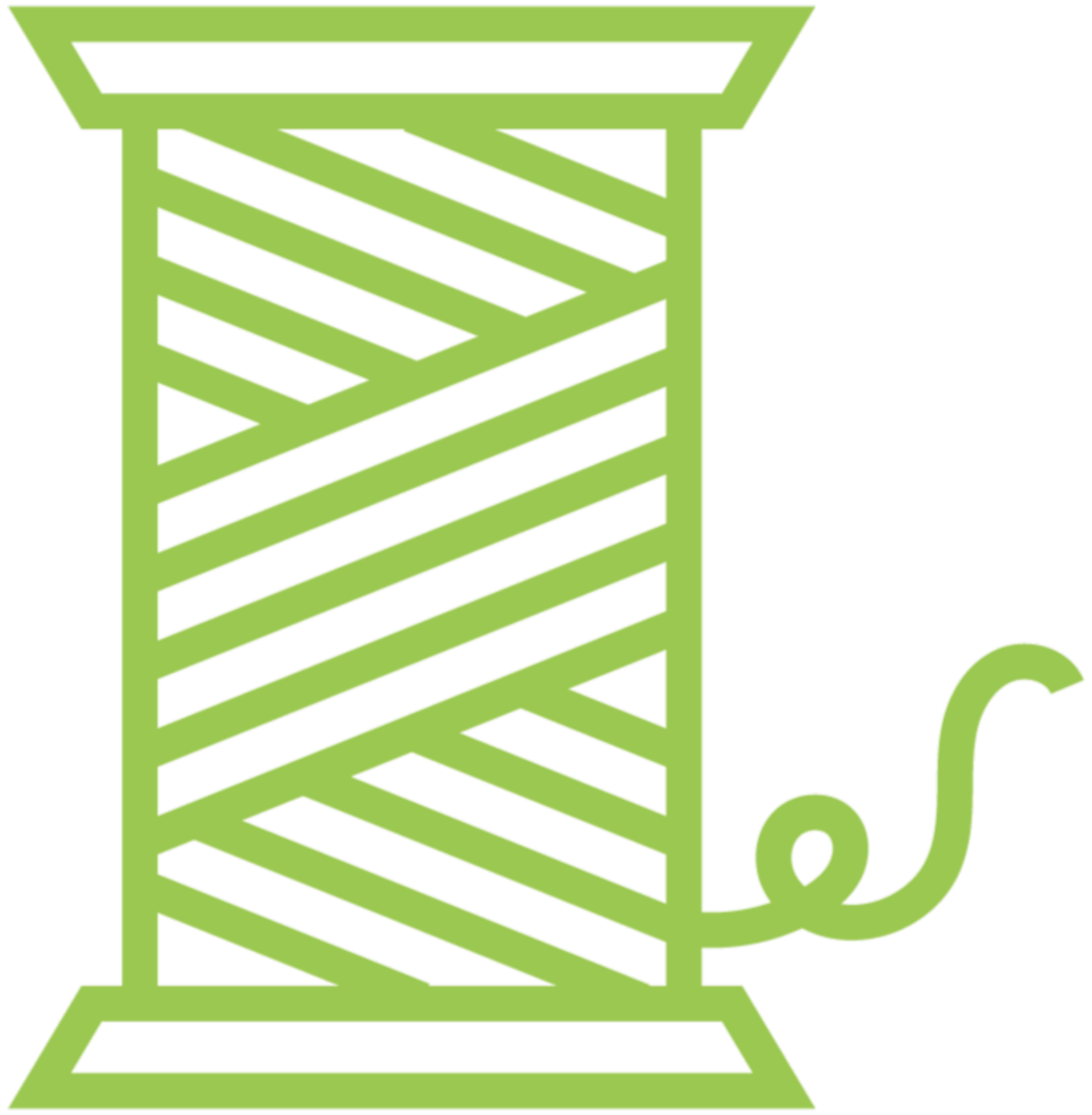


Introduced in .NET Core 3 and C# 8

Supports asynchronous disposal of types

The IServiceProvider supports asynchronous disposal of IAsyncDisposable types it creates

- Until .NET 6, IAsyncDisposable-only types were <u>not</u> supported from scopes and would cause an InvalidOperationException



IServiceScope returned by the “CreateScope” method implements only IDisposable

- The D.I contract could not be changed without breaking third-party containers

.NET 6 solves this:

- Added a new “CreateAsyncScope” extension method
- CreateAsyncScope returns a scope wrapped in an AsyncServiceScope that implements IDisposable and IAsyncDisposable

The wrapper determines if the IServiceScope implementation supports DisposeAsync( ), otherwise falling back to Dispose( )

The built-in container supports scoped IAsyncDisposable services

Demo

Complete initial service registrations for the Tennis Booking application

- Apply the techniques learned so far

The Entity Framework DbContext is registered as a scoped service.

ValidateOnBuild

Configurable on the ServiceProviderOptions

Enabled by default in the development environment

When enabled, a check is performed to verify that all services can be created

Triggered by the call to builder.Build()

?

ASP.NET Core Component Activation

Controllers and Razor pages are activated per-request and are not directly registered with the DI container by default.

Don’t Panic!

Up Next:

Registering More Complex Services