NDC { London }

.NET Core Dependency Injection

The Booster Jab

Steve Collins

#ndclondon
@stevetalkscode

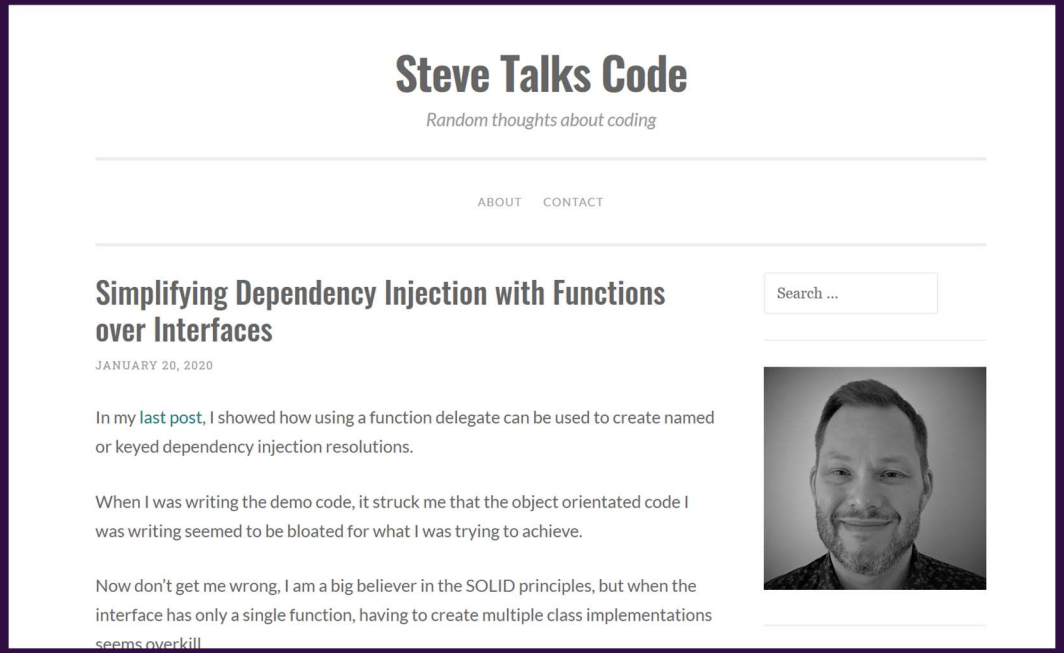
NDC { London }

WHO AM I?

Contract Microsoft stack developer

Blog at https://stevetalkscode.co.uk

@stevetalkscode



The Bits I Assume You Already Know …

NDC { London }

Making the assumption that you already know the basics

- In most cases, will be using a HostBuilder to start the application and initiate creating a container
- An implementation of IServiceCollection is passed to a StartUp class to register services
- An IServiceProvider instance is built from the services registered with the IServiceCollection to create a container
- Out of the box, the Microsoft container only supports Constructor Injection
- ASP.NET Core adds Method Parameter Injection

```
[HttpGet]
public IEnumerable<WeatherForecast> Get([FromServices] ITemperatureConverter tempConverter)
```

- Usually, you need to use other containers for features not supported by the .NET container separately

#ndclondon
@stevetalkscode

The Bits I Assume You Already Know …

NDC { London }

Other injection types not supported out of the box

- Property injection
- Named or keyed service injection
- Convention-based registration

But we have workarounds!

#ndclondon
@stevetalkscode

Integrating with Other DI Containers

NDC { London }

If you need one of these features the following containers may meet your needs

Missing features from .NET Core container

- Property injection
- Injection based on name or key
- Child containers
- Custom lifetime management
- Func<T> support for lazy initialization
- Convention-based registration

Other containers that can be integrated

- Autofac
- Dryloc
- Grace
- LightInject
- Lamar
- Stashbox
- Unity

Usually register one of these other containers with an extension method

The way the external container provider is registered depends on whether .NET Core 2.x or .NET Core 3.x (and now .NET 5)

Have to manage both containers as still have to access the special features via the underlying container

#ndclondon
@stevetalkscode

Integrating with Other DI Containers

NDC { London }

```
// .NET Core 2.x old-style registration of alternative container

public IServiceProvider ConfigureServices(IServiceCollection services)
{
    // Add services using the services collection as normal here
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    // Create the Autofac container and register modules
    var builder = new ContainerBuilder();
    builder.Populate(services);
    builder.RegisterModule(new MyApplicationModule());
    AutofacContainer = builder.Build();

    // Return the Autofac implementation of IServiceProvider
    return new AutofacServiceProvider(AutofacContainer);
}
```

#ndclondon
@stevetalkscode

Registering services with the Microsoft DI Container

NDC { London }

When using a HostBuilder, typically done in the Startup Class

```
public class Startup
{

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton<SomeClassA>();
        services.AddScoped<SomeClassB>();
        services.AddTransient<SomeClassC>();
    }
}
```

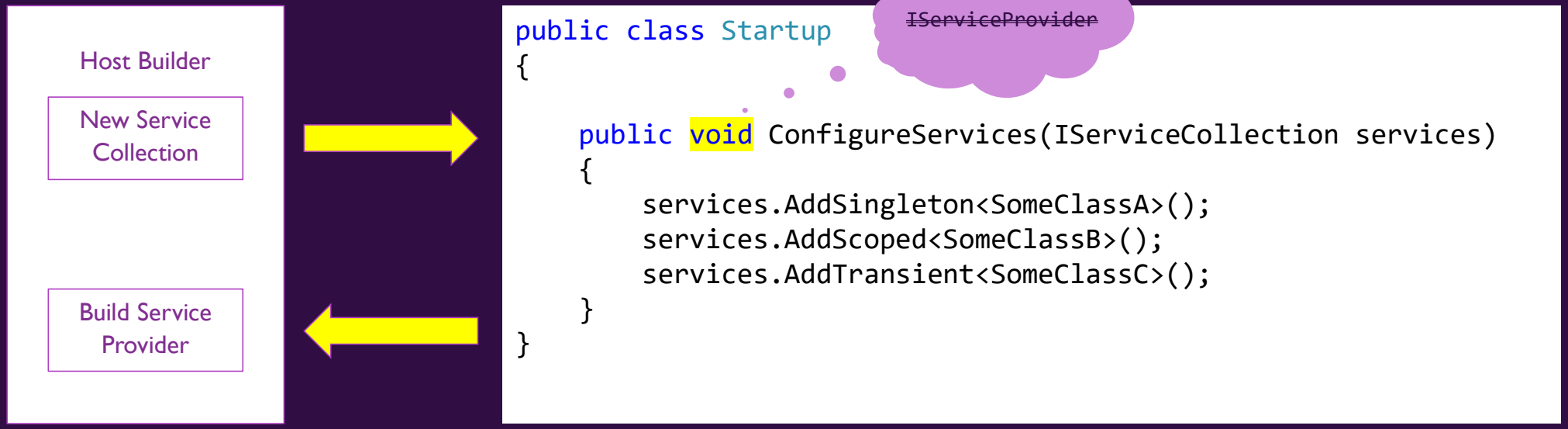
The HostBuilder creates the ServiceCollection instance before passing to the ConfigureServices method in the Startup class

#ndclondon
@stevetalkscode

Registering services with the Microsoft DI Container

NDC { London }

When using a HostBuilder, typically done in the Startup Class



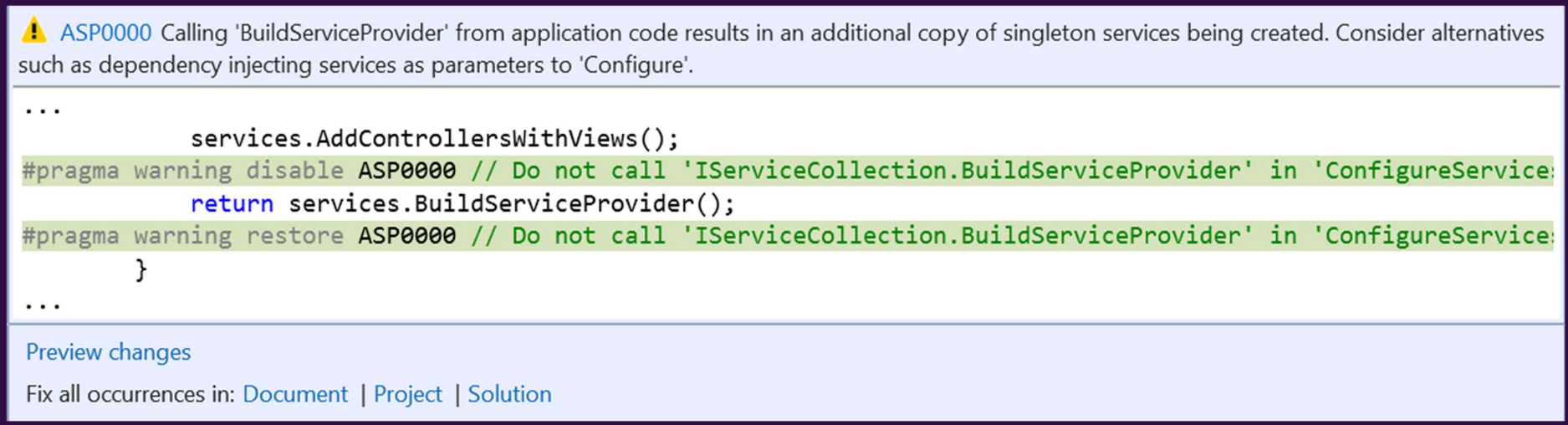
The HostBuilder creates the ServiceCollection instance before passing to the ConfigureServices method in the Startup class

In ASP.NET Core 2.x using the WebHostBuilder it was usual to return IServiceProvider from ConfigureServices, but since .NET Core 3.x, it is not necessary

#ndclondon
@stevetalkscode

Registering services with the Microsoft DI Container

NDC { London }



If using an alternative container, now use the UseServiceProviderFactory extension on the host builder

Then either call

- the ConfigureContainer extension method on the host or
- add a ConfigureContainer method to the StartUp class

#ndclondon
@stevetalkscode

Registering services with the Microsoft DI Container

NDC { London }

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .UseServiceProviderFactory(new AutofacServiceProviderFactory())
            .ConfigureWebHostDefaults(webBuilder => webBuilder.UseStartup<Startup>());
}
```

If using an alternative container, now use the UseServiceProviderFactory extension on the host builder

Then either call

- the ConfigureContainer extension method on the host or
- add a ConfigureContainer method to the StartUp class

#ndclondon
@stevetalkscode

Registering services with the Microsoft DI Container

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .UseServiceProviderFactory(new AutofacServiceProviderFactory())
            .ConfigureWebHostDefaults(webBuilder => webBuilder.UseStartup<Startup>());
            .ConfigureContainer<ContainerBuilder>(cb =>
            {
                // configure the Autofac container here
            });
```

NDC { London }
#ndclondon
@stevetalkscode

Registering services with the Microsoft DI Container

```
public class Startup
{
    public Startup(IConfiguration config) => Configuration = config;
    public IConfiguration Configuration { get;}
    public ILifetimeScope AutofacContainer { get; private set; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
        services.AddOptions();
    }
    public void ConfigureContainer(ContainerBuilder builder) =>
           builder.RegisterModule(new MyApplicationModule());

    public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory) =>
           AutofacContainer = app.ApplicationServices.GetAutofacRoot();
}
```

NDC { London }
#ndclondon
@stevetalkscode

NDC { London }

Service Lifetimes

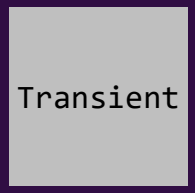
#ndclondon
@stevetalkscode

The Basics of the Microsoft DI Container - Service Lifetimes

NDC { London }

As with most DI containers, it supports three object lifetimes

An instance is created the first time it is requested and remains for the lifetime of the container



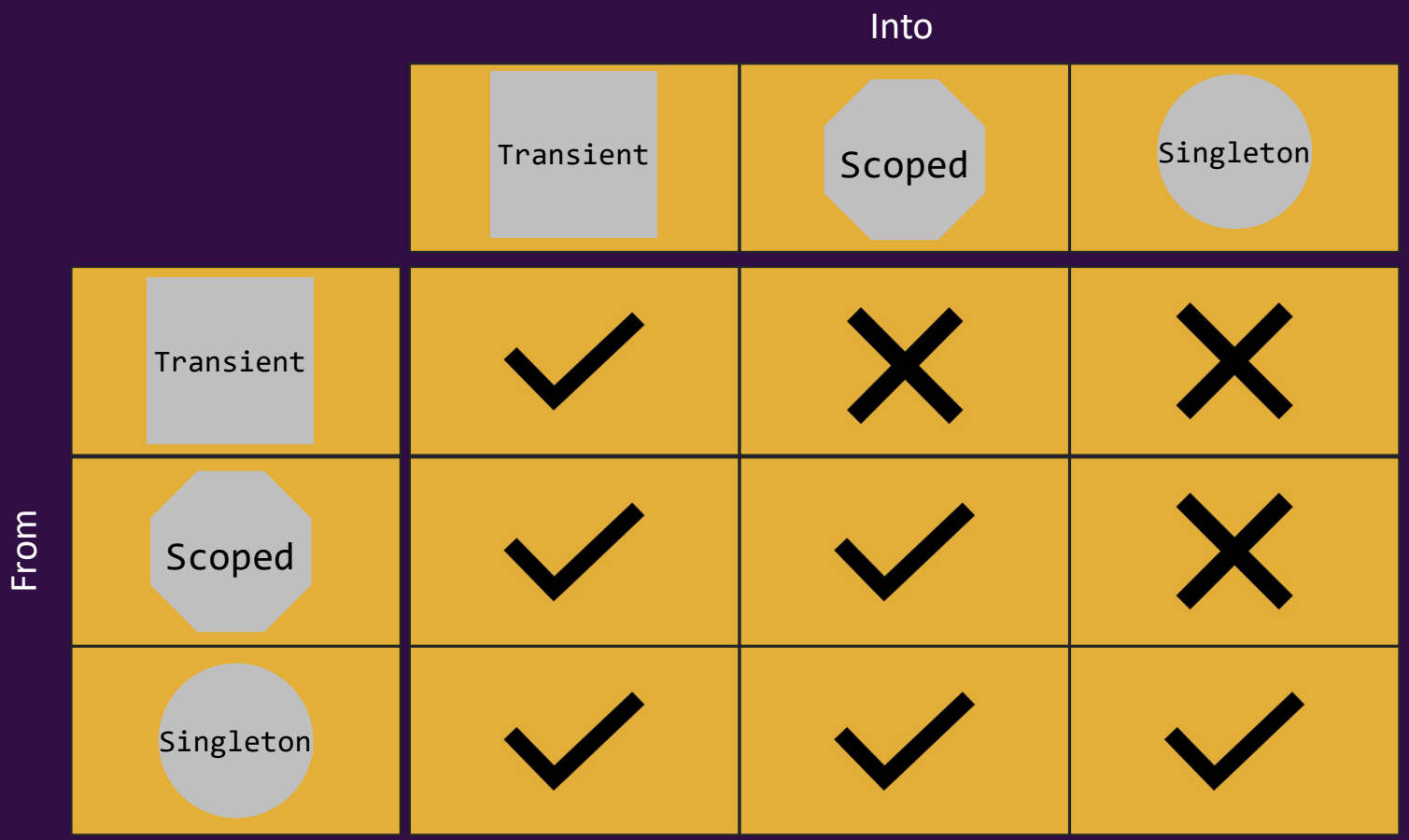
Creates new instance every time a request is made to the container

Hybrid in that a new instance is created for each unit of work, but acts like a singleton for the lifetime of that unit of work

#ndclondon
@stevetalkscode

Scope Compatibility

NDC { London }



#ndclondon
@stevetalkscode

Incompatible Scopes continued

NDC { London }

General misconception that you can't insert incompatible lifetimes into each other

- The Microsoft DI container won't stop you!
- It creates a 'Captured Dependency' where the shorter lifetime object is trapped for the lifetime of the longer lived object

https://www.shutterstock.com/g/darieno79

#ndclondon
@stevetalkscode

Incompatible Scopes continued

NDC { London }

General misconception that you can't insert incompatible lifetimes into each other

- The Microsoft DI container won't stop you!
- It creates a 'Captured Dependency' where the shorter lifetime object is trapped for the lifetime of the longer lived object
- But what about this ?

```
System.InvalidOperationException : Cannot consume scoped service 'AScopedThing' from singleton
'ASingletonThing'.
```

- Happens when ValidateScopes is set to true in the ServiceProviderOptions when BuildServiceProvider is called
- Only happens by default if ASPNETCORE_ENVIRONMENT is set to 'Development'
- Can set it manually for other environments, but there is a performance hit, so the advice is not to do it

#ndclondon
@stevetalkscode

NDC { London }

When to Use a Singleton

#ndclondon
@stevetalkscode

Singletons

NDC { London }

Typical Uses of Singletons in Dependency Injection

- Stateless classes that provide functionality used other classes
- Pooled resources
- In-memory queues
- Factory classes – only need one instance as it is the methods that create instances of the required dependency
- Ambient Singletons - such as the HttpContextAccessor that hide scope complexity by using AsyncLocal to attach to the current Async context and allows injection into other singletons
- Heavily used lookup classes such as read-only caches which do not change through the application lifetime once initialized

#ndclondon
@stevetalkscode

Singletons

NDC { London }

A read-write singleton needs to be made thread-safe as no guarantees who will call it and how it will be called

Give thought to avoiding race conditions

- For collections, use the generic Concurrent collections as these will take care of thread-safety for you
- For read-only properties, ensure that the underlying field is marked as Read Only in code and only set in the constructor / field initialiser
- For read-write properties – ensure that the code includes a thread lock so that only one thread at a time can make an update, or consider AyncLocal storage for performance if there is no inter-dependencies
- If the class implements IDisposable, ensure that this does not leak out to consumers via the container as a consumer could call Dispose() and stop all other consumers from working – consider registering as an interface that does not include Dispose instead of registering the class itself

#ndclondon
@stevetalkscode

Singletons

NDC { London }

System.Collections.Concurrent Namespace

The `System.Collections.Concurrent` namespace provides several thread-safe collection classes that should be used in place of the corresponding types in the System.Collections and System.Collections.Generic namespaces whenever multiple threads are accessing the collection concurrently.

However, access to elements of a collection object through extension methods or through explicit interface implementations are not guaranteed to be thread-safe and may need to be synchronized by the caller.

Classes

BlockingCollection<T>	Provides blocking and bounding capabilities for thread-safe collections that implement IProducerConsumerCollection<T>.
ConcurrentBag<T>	Represents a thread-safe, unordered collection of objects.
ConcurrentDictionary<TKey,TValue>	Represents a thread-safe collection of key/value pairs that can be accessed by multiple threads concurrently.
ConcurrentQueue<T>	Represents a thread-safe first in-first out (FIFO) collection.
ConcurrentStack<T>	Represents a thread-safe last in-first out (LIFO) collection.
OrderablePartitioner<TSource>	Represents a particular manner of splitting an orderable data source into multiple partitions.
Partitioner	Provides common partitioning strategies for arrays, lists, and enumerables.
Partitioner<TSource>	Represents a particular manner of splitting a data source into multiple partitions.

#ndclondon
@stevetalkscode

Singletons

```
public class Account
{
    private readonly object _balanceLock = new object();

    private decimal _balance;

    public Account(decimal initialBalance) => _balance = initialBalance;

    public decimal Debit(decimal amount)
    {
        lock (_balanceLock)
        {
            if (_balance < amount) return 0;
            _balance -= amount;
            return amount;
        }
    }

    public decimal GetBalance()
    {
        lock (_balanceLock)
        {
            return _balance;
        }
    }
}
```

NDC { London }

#ndclondon
@stevetalkscode

Singletons

NDC { London }

```
public interface IDoSomething
 {
     void DoSomething();
 }


 public class DoNotDisposeMe : IDisposable, IDoSomething
 {
     public void DoSomething()
     {
         // implementation ...
     }

     public void Dispose()
     {
        // only gets called by the container, not the consumers if registered
        // as IDoSomething instead of DoNotDisposeMe
     }
 }
```

Hidden from Interface consumers but not the container

#ndclondon
@stevetalkscode

Transients

NDC { London }

The simplest of the three lifetimes

- New instance created every time requested from the container
- Best for lightweight, short-lived objects
- Can accept singletons and scoped instances in constructor
- Be careful of doing any 'heavy' work in the constructor
- If implementing IDisposable, the container will look after this

#ndclondon
@stevetalkscode

NDC { London }

Scoped Dependencies

APPROACH WITH CAUTION!

#ndclondon
@stevetalkscode

Understanding Scoped Dependencies

NDC { London }

Stands in the middle of the road between transients and singletons

- Should only be accessed from the scope container
- ASP.NET Core creates a new scoped container accessible via the HttpContext. RequestServices property
- Can manually create a scope by using the CreateScope extension method on the IServiceProvider to create an instance of IServiceScope
    - This actually resolves IServiceScopeFactory from the ServiceProvider and calls CreateScope on that
    - Note the IServiceScope needs to be disposed when finished with (IDisposable)
- An IServiceScope has a single property of ServiceProvider which is an IServiceProvider that can serve both scoped dependencies from itself and transient/singleton entities from the root service provider
- If implementing IDisposable, the container looks after the disposal of the object if it created the instance

#ndclondon
@stevetalkscode

Understanding Scoped Dependencies

NDC { London }

```
namespace Microsoft.AspNetCore.Http.Features
{
    public class RequestServicesFeature : IServiceProvidersFeature, IDisposable, IAsyncDisposable
    {
        public RequestServicesFeature(HttpContext context, IServiceScopeFactory? scopeFactory)
        {
            _context = context;
            _scopeFactory = scopeFactory;
        }

        public IServiceProvider RequestServices
        {
            get
            {
                if (!_requestServicesSet && _scopeFactory != null)
                {
                    _context.Response.RegisterForDisposeAsync(this);
                    _scope = _scopeFactory.CreateScope();
                    _requestServices = _scope.ServiceProvider;
                    _requestServicesSet = true;
                }
                return _requestServices!;
            }

            set
            {
                _requestServices = value;
                _requestServicesSet = true;
            }
        }
    }
}
```

#ndclondon
@stevetalkscode

Understanding Scoped Dependencies

NDC { London }

If injecting scoped dependencies into convention-based middleware, don't inject via constructor – add to the Invoke/InvokeAsync methods which can have additional parameters added to the signature

```
public class MyCustomConventionalMiddleware
{
    private readonly RequestDelegate _next;

    public MyCustomConventionalMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context, IMyScopedDependency dependency)
    {
        … can do some work with the scoped dependency here …

        await _next(context);

        … or here!

    }}
```

DON'T inject here!

Otherwise dependency is captured

DO inject here –in pipeline so scoped dependency is correctly resolved

#ndclondon
@stevetalkscode

Understanding Scoped Dependencies

NDC { London }

If injecting scoped dependencies into convention-based middleware, don't inject via constructor – add to the Invoke/InvokeAsync methods which can have additional parameters added to the signature

```
public class MyCustomConventionalMiddleware
{
    private readonly RequestDelegate _next;

    public MyCustomConventionalMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context, IMyScopedDependency dependency)
    {
        … can do some work with the scoped dependency here …

        await _next(context);

        … or here!

    }}
```

Gets called by reflection to create the instance

Delegate to method is inserted into pipeline to call this

```
// This method gets called by the runtime. Use this method to configur the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IWebHostEnvironment env
{
    app.UseRouting();
    app.UseMiddleware<MyCustomConventionalMiddleware>();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

#ndclondon
@stevetalkscode

Understanding Scoped Dependencies

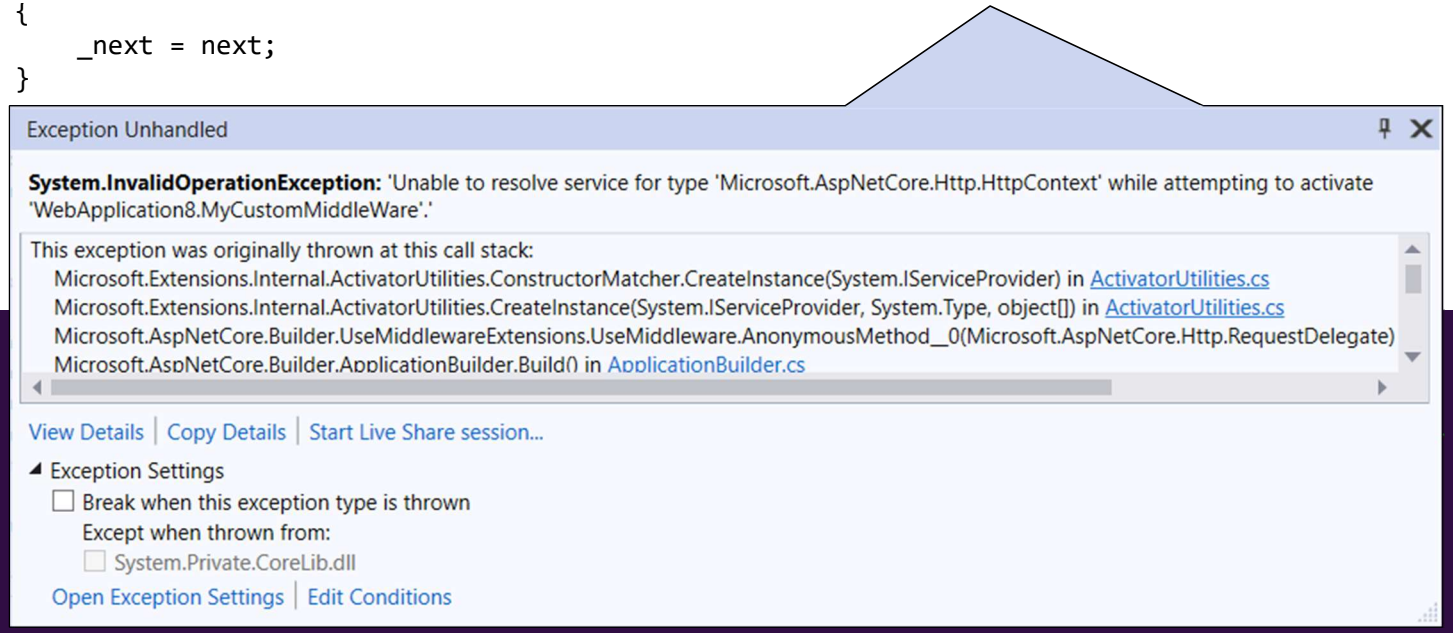
NDC { London }

If injecting scoped dependencies into convention-based middleware, don't inject via constructor – add to the Invoke/InvokeAsync methods which can have additional parameters added to the signature

```
public class MyCustomConventionalMiddleware
{
    private readonly RequestDelegate _next;

    public MyCustomConventionalMiddleware(RequestDelegate next, HttpContext context)
    {
        _next = next;
    }
}
```

Gets called by reflection to create the instance



#ndclondon
@stevetalkscode

Understanding Scoped Dependencies

NDC { London }

If injecting scoped dependencies into convention-based middleware, don't inject via constructor – add to the Invoke/InvokeAsync methods which can have additional parameters added to the signature

```
public class MyCustomConventionalMiddleware
{
    private readonly RequestDelegate _next;

    public MyCustomConventionalMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context, IMyScopedDependency dependency)
    {
        … can do some work with the scoped dependency here …

        await _next(context);

        … or here!

    }
}
```

In pipeline so scoped dependency is correctly resolved from the scoped provider

Invokes the next middleware in the pipeline

#ndclondon
@stevetalkscode

Understanding Scoped Dependencies

NDC { London }

Factory-based middleware works differently – it is created on each request via Dependency Injection and the rules for injection are reversed

```
public class MyCustomFactoryMiddleware : IMiddleware
{

    private readonly IMyScopedDependency _dependency;

    public MyCustomFactoryMiddleware(IMyScopedDependency dependency)
    {
        _dependency = dependency;
    }

    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
          … can do some work with the scoped dependency here …

        await next(context);

        … or here!      }
}
```

Implements IMiddleware

Inject dependencies into constructor

Interface signature is fixed

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/extensibility

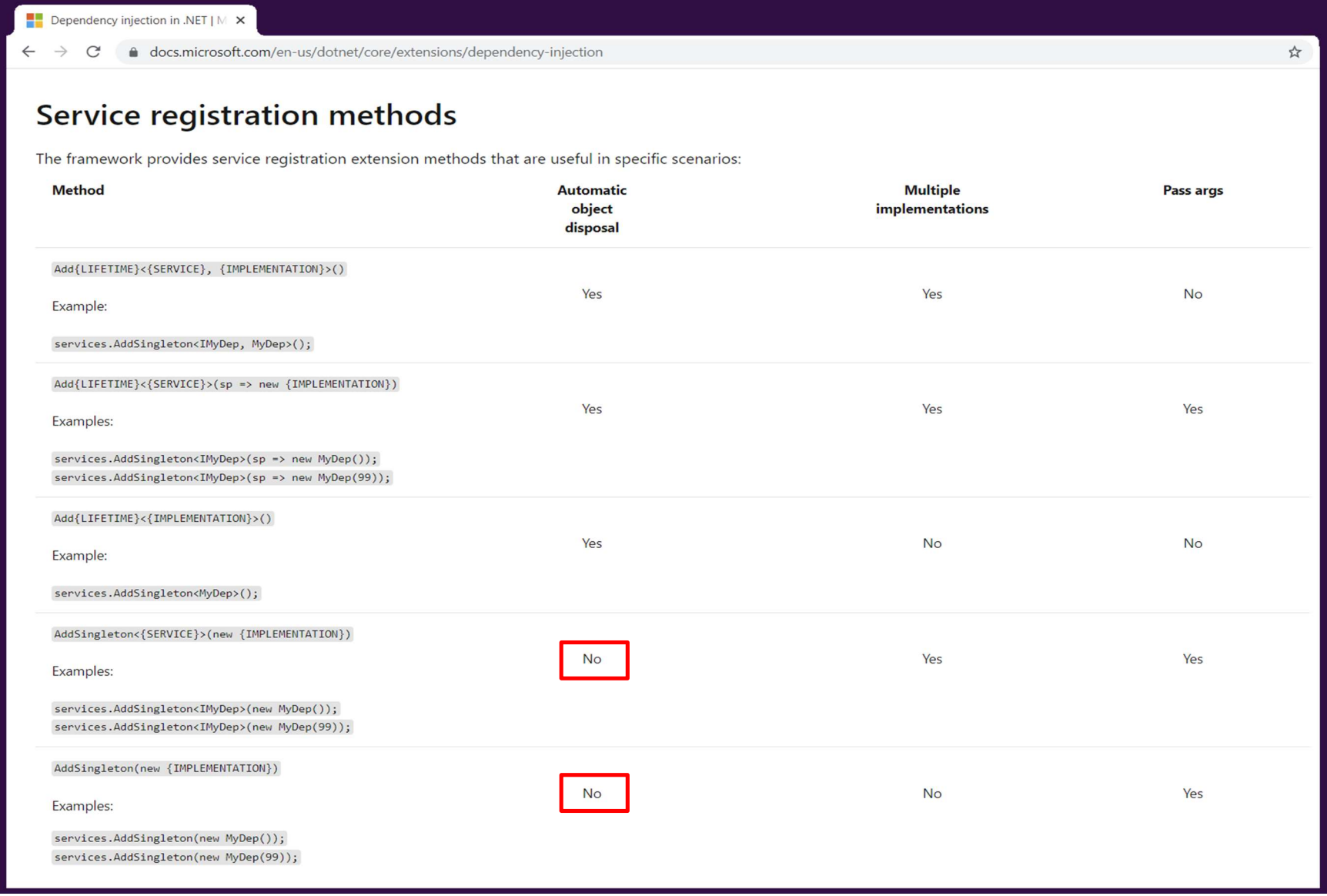
#ndclondon
@stevetalkscode

NDC { London }

Handling Disposables

#ndclondon
@stevetalkscode

Disposables and Memory Leaks



Service registration methods

The framework provides service registration extension methods that are useful in specific scenarios:

Method	Automatic object disposal	Multiple implementations	Pass args

`Add{LIFETIME}<{SERVICE}, {IMPLEMENTATION}>()`
Example:
`services.AddSingleton<IMyDep, MyDep>();`
	Yes	Yes	No

`Add{LIFETIME}<{SERVICE}>(sp => new {IMPLEMENTATION})`
Examples:
`services.AddSingleton<IMyDep>(sp => new MyDep());`
`services.AddSingleton<IMyDep>(sp => new MyDep(99));`
	Yes	Yes	Yes

`Add{LIFETIME}<{IMPLEMENTATION}>()`
Example:
`services.AddSingleton<MyDep>();`
	Yes	No	No

`AddSingleton<{SERVICE}>(new {IMPLEMENTATION})`
Examples:
`services.AddSingleton<IMyDep>(new MyDep());`
`services.AddSingleton<IMyDep>(new MyDep(99));`
	No	Yes	Yes

`AddSingleton(new {IMPLEMENTATION})`
Examples:
`services.AddSingleton(new MyDep());`
`services.AddSingleton(new MyDep(99));`
	No	No	Yes

NDC { London }

#ndclondon
@stevetalkscode

Disposables and Memory Leaks

NDC { London }

If creating disposable instances in the StartUp class

- Inject IHostApplicationLifetime instance into the Configure method
- Register a shutdown handler
- In the handler, dispose the object

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllersWithViews();
        services.AddSingleton<IMyDisposableThing>(new MyDisposableThing(12345));
    }

    public void Configure(IApplicationBuilder app, IHostApplicationLifetime applicationLifetime,
                          IMyDisposableThing disposableThing)
    {
        applicationLifetime.ApplicationStopping.Register(OnApplicationShutdown, disposableThing);
    }

    private void OnApplicationShutdown(object thingToDispose)
    {
        (thingToDispose as IDisposable)?.Dispose();
    }
}
```

#ndclondon
@stevetalkscode

Disposables and Memory Leaks

NDC { London }

The DI container will take care of disposing IDisposable objects, but

- Will only do so if the container itself created the object
- A created object must still take care of disposing objects it creates itself
- Thought needs to be given to the disposable of objects injected into the constructor
- Avoid exposing the IDisposable interface to consumers of the container
    - Consider extracting a restricted interface (if it is your code)
    - If not your code, consider a Façade, Adapter, Bridge or Proxy class to hide the underlying code then expose a restricted interface on the wrapper (that also must implement IDisposable and dispose of the inner object)
- Be wary of 'control freak' classes that dispose of instances when it is not their responsibility – this is usually sign of a bad design and a misplaced responsibility

#ndclondon
@stevetalkscode

NDC { London }

Common Mistakes

#ndclondon
@stevetalkscode

Accidentally Multi-registering Dependencies

NDC { London }

The DI container allows you to register the same service type multiple times

- This may be deliberate to register different implementations

```
services.AddSingleton<IMove, Walk>();
services.AddSingleton<IMove, Jog>();
services.AddSingleton<IMove, Run>();
```

- May be due to a code merge that ends up with the same registration twice
- May be unexpected due to extension methods already having registered a service type
- This may have unintended consequences

#ndclondon
@stevetalkscode

Last In Wins Principle

NDC { London }

If a consumer requires a dependency to be injected, and there are multiple registrations, the last to be registered is the instance that is returned to the consumer

- If an extension method is used after your code and registers a service type, you may get unintended behavior in your application if the instance is different to that expected

- The lifetime of the last entry may be inappropriate to the lifetime being injected into (E.g. Scoped being injected into a Singleton)

The multiple registration may be deliberate as you may want to inject a collection of instances

- If this is the case, the consumer constructor should have a parameter of IEnumerable<*ServiceType*> so it can iterate over the collection

#ndclondon
@stevetalkscode

Avoiding Accidentally Multi-registering Dependencies

NDC { London }

To avoid accidentally registering services twice

Use the TryAdd* extension methods

```
services.TryAddSingleton<IPlanet, Earth>();      ✓
services.TryAddSingleton<IPlanet, Mars>();       ✗
services.TryAddSingleton<IPlanet, Jupiter>();    ✗
services.TryAddSingleton<IPlanet, Earth>();      ✗
services.TryAddSingleton<IPlanet, Mars>();       ✗
services.TryAddSingleton<IPlanet, Jupiter>();    ✗
```

First Entry Wins

1

or the TryAddEnumerable extension

```
services.TryAddEnumerable(ServiceDescriptor.Describe(typeof(IPlanet), typeof(Earth), ServiceLifetime.Singleton));      ✓
services.TryAddEnumerable(ServiceDescriptor.Describe(typeof(IPlanet), typeof(Mars), ServiceLifetime.Singleton));       ✓
services.TryAddEnumerable(ServiceDescriptor.Describe(typeof(IPlanet), typeof(Jupiter), ServiceLifetime.Singleton));    ✓

services.TryAddEnumerable(ServiceDescriptor.Describe(typeof(IPlanet), typeof(Earth), ServiceLifetime.Singleton));      ✗
services.TryAddEnumerable(ServiceDescriptor.Describe(typeof(IPlanet), typeof(Mars), ServiceLifetime.Singleton));       ✗
services.TryAddEnumerable(ServiceDescriptor.Describe(typeof(IPlanet), typeof(Jupiter), ServiceLifetime.Singleton));    ✗
```

First Entry Per Service + Implementation Type Wins

3

#ndclondon
@stevetalkscode

NDC { London }

Handy Tips

#ndclondon
@stevetalkscode

Deliberate Multiple Registrations of the Same Interface

NDC { London }

May want to register multiple implementations of the same service type

- A GoF Strategy or Visitor pattern where multiple process need to be applied in order
- Example may be Validation Rules that are applied one at a time over a request
- Consider implementing IComparable<T> if the implementations can be ordered based on properties
- Possibly create a class that inherits from IEnumerable<T> and use Yield to fix the order, then inject this class into the consumer
- Could do something similar directly inside the StartUp registrations with a lambda expression

```
services.AddSingleton<IBackpack, Red>();
services.AddSingleton<IBackpack, Cyan>();
services.AddSingleton<IBackpack, Navy>();
…
var servicesFound = serviceProvider.GetRequiredService<IEnumerable<IBackpack>>();
```

#ndclondon
@stevetalkscode

Deliberate Multiple Registrations of the Same Interface

NDC { London }

May want to register multiple implementations of the same service type

- A GoF Strategy or Visitor pattern where multiple process need to be applied in order
- Example may be Validation Rules that are applied one at a time over a request
- Consider implementing IComparable<T> if the implementations can be ordered based on properties
- Possibly create a class that inherits from IEnumerable<T> and use Yield to fix the order, then inject this class into the consumer
- Could do something similar directly inside the StartUp registrations with a lambda expression

```
services.AddSingleton<IBackpack, Red>();
services.AddSingleton<IBackpack, Cyan>();
services.AddSingleton<IBackpack, Navy>();
…
var servicesFound = serviceProvider.GetRequiredService<IEnumerable<IBackpack>>();
```

#ndclondon
@stevetalkscode

Same Class, Many Interfaces

NDC { London }

A class may implement multiple interfaces

- If each is registered in the usual manner, you get a new instance per registration when requested by a consumer
- If there is not one already, consider creating an aggregate interface that implements all the other interfaces
- Have to do some redirection to all point to same instance registered against the aggregate interface, but allows requested via smaller interfaces (ISP in SOLID)

```
public interface IMyInterfaceAggregate : IMyInterface1, IMyInterface2, IMyInterface3
```

```
public class MyImplementation : IMyInterfaceAggregate : IDisposable
```

```
services.AddSingleton<IMyInterfaceAggregate, MyImplementation>();

services.AddSingleton<IMyInterface1>(sp => sp.GetRequiredService<IMyInterfaceAggregate>());
services.AddSingleton<IMyInterface2>(sp => sp.GetRequiredService<IMyInterfaceAggregate>());
services.AddSingleton<IMyInterface3>(sp => sp.GetRequiredService<IMyInterfaceAggregate>());
```

#ndclondon
@stevetalkscode

Registering Open Generics

NDC { London }

Allow the consumer to decide the generic type it wants

- An open generic does not specify the generic type itself. E.g. List<T>
- Cannot use generic versions of extension methods to register an open generic, so you must use the appropriate non-generic methods to register

```
services.AddSingleton(typeof(IDoSomethingGeneric<>), typeof(DoSomethingGeneric<>))
```

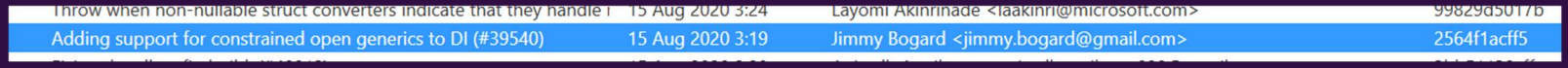
- Commonly seen in ASP.NET Core with the AddLogging extension method has been used so that you can request ILogger<MyType> in the constructor of your controllers
- Another commonly used example is AddOptions extension which registers
  - IOptions<T>
  - IOptionsSnapshot<T>
  - IOpionsMonitor<T>
- These are used when you have used Configure<T> to bind configuration to a class

#ndclondon
@stevetalkscode

Registering Constrained Generics

NDC { London }

New in .NET 5 thanks to pull request from Jimmy Bogard



- Previously resolving collections of open generics based on a generic constraint threw an exception
- By adding a try-catch-swallow, constrained generics now work

- Read Jimmy's blog post https://jimmybogard.com/constrained-open-generics-support-merged-in-net-core-di-container/
- Shows why the DI container does not change between versions of .NET

#ndclondon
@stevetalkscode

NDC { London }

Design Patterns

#ndclondon
@stevetalkscode

Factory and Builder Patterns

NDC { London }

Sometimes, it is not possible to rely on the container to do all the work of creating the instance

- Not all the parameters for the constructor can be created from the container
- Whilst some of the parameters can be created, they need to be configured before being consumer by the constructor
- You want to be in control of disposing the created class rather than leaving to the container
- You want to use property injection, but the container does not support it

#ndclondon
@stevetalkscode

Factory and Builder Patterns

NDC { London }

```
public class SomeLongLivedThing
{
    public string Message { get; } = "I'm a long lived thing!";
}
```

```
public class SomeShortLivedThing
{
    public SomeShortLivedThing(
        SomeLongLivedThing coreThing,
        string name)
    {
        CoreThing = coreThing;
        Name = name;
        UniqueIdentity = Guid.NewGuid();
    }

    public SomeLongLivedThing CoreThing { get; }
    public string Name { get; }
    public Guid UniqueIdentity { get; }
}
```

```
[ApiController]
[Route("[controller]")]
public class ThingController
{
    private readonly ShortLivedThingFactory _factory;
    public ThingController(ShortLivedThingFactory factory)
    {
        _factory = factory;
    }

    [HttpGet]
    public SomeShortLivedThing Get() =>
      _factory.
          MakeShortLivedThing($"Hello from {nameof(ThingController)}");}
```

```
public class ShortLivedThingFactory
{
    private readonly SomeLongLivedThing _coreThing;
    public ShortLivedThingFactory(SomeLongLivedThing coreThing) => _coreThing = coreThing;
    public SomeShortLivedThing MakeShortLivedThing(string name) => new (_coreThing, name);
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<SomeLongLivedThing>();
    services.AddSingleton<ShortLivedThingFactory>();
}
```

#ndclondon
@stevetalkscode

Factory and Builder Patterns

NDC { London }

The Gang of Four (*GoF*) Factory design pattern can be of help here

- Usually, a factory will be registered as a singleton
- If it needs to resolve singleton dependencies, pass these into the constructor
- For required transients, make IServiceProvider a constructor parameter
- Runtime and scoped dependencies should be passed as parameters to a Create method
- The Create method uses all these inputs to create a new instance
- If the created object needs properties set before returning the instance, it can be done here (effectively providing a workaround for the lack of property injection)
- If the new object implements IDisposable, it is up to the consumer to dispose of it as the container did not create it

#ndclondon
@stevetalkscode

Factory and Builder Patterns

NDC { London }

The GoF Builder pattern is useful for adding data to a builder object then using the Build method to create the final object.

Factory – fixed options

Builder – flexible options

#ndclondon
@stevetalkscode

Factory and Builder Patterns

NDC { London }

The GoF Builder pattern is useful for adding data to a builder object then using the Build method to create the final object.

- Register a builder with the container as a transient.
- Should not register as a singleton as manipulating state, but can expose through a factory (registered as a singleton) to create a new instance on demand if needed by other singletons to avoid captured dependency
- In the consuming class, request the builder in the constructor
- Consumer calls methods or sets properties to define how it wants the instance configured
- Consumer calls Build method to create the instance (usually immutable from that point)
- If created class implements IDisposable, the consumer needs to dispose of created object when finished with (as not created by the container, so no auto-disposal)

#ndclondon
@stevetalkscode

NDC { London }

Working Around Limitations

#ndclondon
@stevetalkscode

Unable to Register Value Types

NDC { London }

```
public class StructDemo
{
  [Fact]
  public void TestValueTypeConstruction()
  {
    var services = new ServiceCollection();
    services.AddTransient<MyValueType>();
    var serviceProvider = services.BuildServiceProvider(true);
    var result =
          serviceProvider.GetRequiredService<MyValueType>();
    Assert.IsType<MyValueType>(result);
  }

  public struct MyValueType
  {
    public string FirstName;
    public string LastName;
  }
}
```

struct RecordFactoryDemo.StructDemo.MyValueType

CS0452: The type 'StructDemo.MyValueType' must be a reference type in order to use it as parameter 'TService' in the generic type or method 'ServiceCollectionServiceExtensions.AddTransient<TService>(IServiceCollection)'

The type 'RecordFactoryDemo.StructDemo.MyValueType' must be a reference type in order to use it as parameter 'TService'

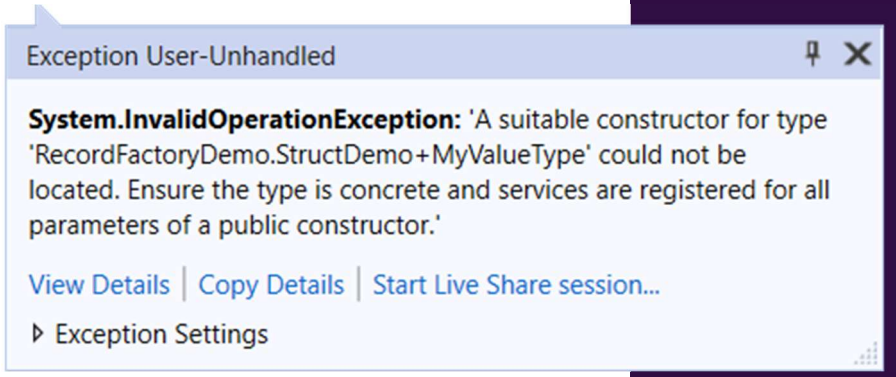
Show potential fixes (Ctrl+.)

#ndclondon
@stevetalkscode

Unable to Register Value Types

```
public class StructDemo
{
  [Fact]
  public void TestValueTypeConstruction()
  {
    var services = new ServiceCollection();
    services.AddTransient(typeof(MyValueType));
    var serviceProvider = services.BuildServiceProvider(true);
    var result =
          serviceProvider.GetRequiredService<MyValueType>();
    Assert.IsType<MyValueType>(result);
  }

  public struct MyValueType
  {
    public string FirstName;
    public string LastName;
  }
}
```



Unable to Register Value Types

NDC { London }

```
[Fact]
public void TestValueTypeConstruction()
{
  var services = new ServiceCollection();
  services.AddSingleton<MyValueTypeFactory>();
  var serviceProvider = services.BuildServiceProvider(true);
  var factory =
       serviceProvider.GetRequiredService<MyValueTypeFactory>();
  var result = factory.Create("Steve", "TalksCode");
  Assert.IsType<MyValueType>(result);
}

public class MyValueTypeFactory
{
  public MyValueType Create(string firstName, string lastName)
    => new MyValueType {FirstName = firstName, LastName = lastName};
}
```

#ndclondon
@stevetalkscode

Registering C# 9 Records

NDC { London }

- Work like value types (value equality), but are reference types like classes
- Less boiler plate code (equality, deconstructors,
- Can register in same way as classes
- More likely to create using Factory Pattern as likely to have some form of user sourced input parameters
- Blogged : http://stevetalkscode.co.uk/c-sharp-9-record-factories

#ndclondon
@stevetalkscode

Using Delegates Instead of Interfaces

NDC { London }

The DI container is not limited to just interfaces and regular classes

- Can also register Delegate types
- Delegates can be useful for single method implementations instead of going through the process of creating classes and interfaces
- Good way of making static function injectable (and therefore can be mocked in a unit test)
- See my blog post at https://stevetalkscode.co.uk/simplifying-di-with-functions

```
public delegate DateTime GetCurrentUtcDateTime();

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllersWithViews();
        services.AddSingleton<GetCurrentUtcDateTime>(sp =>  () => DateTime.UtcNow);
    }
}
```

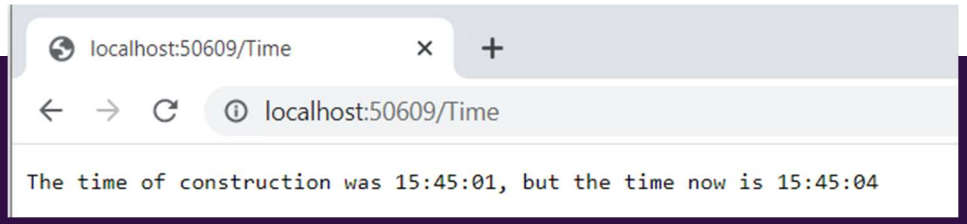
#ndclondon
@stevetalkscode

Using Delegates Instead of Interfaces

NDC { London }

```
[ApiController]
[Route("[controller]")]
public class TimeController
{
    private readonly GetCurrentUtcDateTime _timeDelegate;
    private readonly DateTime _timeAtConstruction;
    public TimeController(GetCurrentUtcDateTime timeDelegate)
    {
        _timeDelegate = timeDelegate;
        _timeAtConstruction = timeDelegate();
    }

    [HttpGet]
    public async Task<string> Get()
    {
        await Task.Delay(3000).ConfigureAwait(false);
        return $"The time of construction was {_timeAtConstruction.ToLongTimeString()}"
               + $", but the time now is {_timeDelegate().ToLongTimeString()}";
    }
}
```



#ndclondon
@stevetalkscode

Injecting Dependencies by Name or Key

NDC { London }

The Microsoft DI container does NOT support this, but other containers do

- Considered to be an anti-pattern, so why do this?
- Can manually add this functionality by using a delegate registered with the container to match up an existing registration with a name as a string or some other key such as an enumeration
- Either requires some hard coding of names or create a dictionary of names/keys to functions that call the service provider to create the instance required
- See my blog post at https://stevetalkscode.co.uk/named-dependencies-part-2 for details

#ndclondon
@stevetalkscode

Injecting Dependencies by Name or Key

NDC { London }

```
private interface IKelvinMapper
{
    decimal ToKelvins(decimal value);
}

private class CentigradeToKelvinMapper : IKelvinMapper
{
    public decimal ToKelvins(decimal value) => value + 273.15M;
}

private class FahrenheitToKelvinMapper : IKelvinMapper
{
    public decimal ToKelvins(decimal value) => (value + 459.67M) * 5 / 9;
}

private class RankineToKelvinMapper : IKelvinMapper
{
    public decimal ToKelvins(decimal value) => value * 5 / 9;
}

private class KelvinToKelvinMapper : IKelvinMapper
{
    public decimal ToKelvins(decimal value) => value;
}
```

#ndclondon
@stevetalkscode

Injecting Dependencies by Name or Key

```
public delegate IKelvinMapper KelvinConverterMapper(string key);

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton<CentigradeToKelvinMapper>();
        services.AddSingleton<FahrenheitToKelvinMapper>();
        services.AddSingleton<RankineToKelvinMapper>();
        services.AddSingleton<KelvinToKelvinMapper>();
        services.AddSingleton<KelvinConverterMapper>(provider => key =>
        {
            switch ((string.IsNullOrEmpty(key) ? " " : key).ToUpper()[0])
            {
                case 'C':
                    return provider.GetRequiredService<CentigradeToKelvinMapper>();

                case 'F':
                    return provider.GetRequiredService<FahrenheitToKelvinMapper>();

                case 'R':
                    return provider.GetRequiredService<RankineToKelvinMapper>();

                case 'K':
                    return provider.GetRequiredService<KelvinToKelvinMapper>();
                default:
                    return null;
            }
        });
    }
}
```

NDC { London }
#ndclondon
@stevetalkscode

Injecting Dependencies by Name or Key

NDC { London }

```
public class GetTemperatures
{
    private KelvinConverterMapper _mapper;

    public GetTemperatures(KelvinConverterMapper mapper)
    {
        _mapper = mapper;
    }

    public decimal? GetTemperature(string scale, decimal value) =>
                        _mapper(scale)?.ToKelvins(value);
}
```

#ndclondon
@stevetalkscode

Using Decorator Classes to Add Functionality

NDC { London }

A decorator is another GoF pattern

- It extends or alters the functionality of objects at run-time by wrapping them in an object of a decorator class as an alternative to using inheritance to
- Can use to add functionality like logging over the top of an existing class
- Take the original class in the constructor parameters
- Implement the same interface
- Registering the original using the class type as the service type
- Register the decorator using the interface as the service type

#ndclondon
@stevetalkscode

Using Decorator Classes to Add Functionality

```
public interface IDoSomething
{
    string GetHelloMessage(string name);
}
```

```
public class DoSomething : IDoSomething
{
    public string GetHelloMessage
                            (string name)
    {
        return $"Hello {name}";
    }
}
```

```
public class Startup
{
   public void ConfigureServices(IServiceCollection services)
   {
     services.AddLogging()
     services.AddSingleton<DoSomething>();
     services.AddSingleton<IDoSomething, LoggingDoSomething>()
   }
}
```

```
public class LoggingDoSomething : IDoSomething
{
    private DoSomething _original;
    private ILogger<DoSomething> _logger;

    public LoggingDoSomething(DoSomething original, ILogger<DoSomething> logger)
    {
        _original = original;
        _logger = logger;
    }

    public string GetHelloMessage(string name)
    {
        _logger.LogInformation($"name received is '{name}'");
        var retVal = _original.GetHelloMessage(name);
        _logger.LogInformation($"Outgoing message is '{retVal}'");
        return retVal;
    }
}
```

NDC { London }
#ndclondon
@stevetalkscode

Searching for Dependencies with Third Party Libraries

NDC { London }

Service registration can become laborious if there are lots of classes to register

Can use libraries that scan assemblies for classes that implement interfaces and perform the registration for you

- Scrutor – probably the most popular library of this kind
  - Also handles registering decorator classes
  - https://github.com/khellang/Scrutor

- If using a third-party container, many of these support assembly scanning

Personally, the control freak in me likes to do this all myself, but these libraries may be of help if you have a massive number of classes to register

#ndclondon
@stevetalkscode

Warming up Services Before Use

NDC { London }

Some services can be slow to instantiate on first use

- Consider having a background task that runs at startup to 'warm up' instantiation of objects
  - Singletons get instantiated and initialized at startup instead of first 'real' usage so ready-to-run on that first 'real' call
  - Transients and scoped instances benefit from JIT compilation
  - https://andrewlock.net/reducing-latency-by-pre-building-singletons-in-asp-net-core/

#ndclondon
@stevetalkscode

Thank You

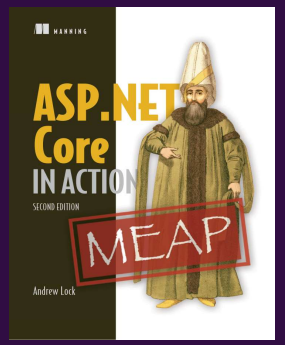
NDC { London }

From me

- @stevetalkscode
- Blog https://stevetalkscode.co.uk

Andrew Lock

- ASP.NET Core in Action (Manning)– second edition available as EAP
- https://andrewlock.net/

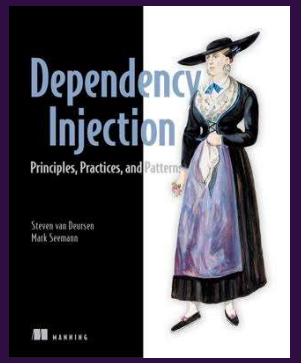


Microsoft Docs

- https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection

Mark Seemann

- https://blog.ploeh.dk/
- Dependency Injection Principle, Practices and Patterns (Manning)



#ndclondon
@stevetalkscode