.NET dependency injection

Article • 07/19/2023

.NET supports the dependency injection (DI) software design pattern, which is a technique for achieving Inversion of Control (IoC) between classes and their dependencies. Dependency injection in .NET is a built-in part of the framework, along with configuration, logging, and the options pattern.

A *dependency* is an object that another object depends on. Examine the following `MessageWriter` class with a `Write` method that other classes depend on:

C#

```
public class MessageWriter
{
    public void Write(string message)
    {
        Console.WriteLine($"MessageWriter.Write(message: \"{mes-
sage}\")");
    }
}
```

A class can create an instance of the `MessageWriter` class to make use of its `Write` method. In the following example, the `MessageWriter` class is a dependency of the `Worker` class:

C#

```
public class Worker : BackgroundService
{
    private readonly MessageWriter _messageWriter = new
MessageWriter();

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            _messageWriter.Write($"Worker running at:
{DateTimeOffset.Now}");
            await Task.Delay(1000, stoppingToken);
        }
    }
}
```

The class creates and directly depends on the `MessageWriter` class. Hard-coded dependencies, such as in the previous example, are problematic and should be avoided for the following reasons:

- To replace `MessageWriter` with a different implementation, the `Worker` class must be modified.
- If `MessageWriter` has dependencies, they must also be configured by the `Worker` class. In a large project with multiple classes depending on `MessageWriter`, the configuration code becomes scattered across the app.
- This implementation is difficult to unit test. The app should use a mock or stub `MessageWriter` class, which isn't possible with this approach.

Dependency injection addresses these problems through:

- The use of an interface or base class to abstract the dependency implementation.
- Registration of the dependency in a service container. .NET provides a built-in service container, IServiceProvider. Services are typically registered at the app's start-up and appended to an IServiceCollection. Once all services are added, you use BuildServiceProvider to create the service container.
- *Injection* of the service into the constructor of the class where it's used. The framework takes on the responsibility of creating an instance of the dependency and disposing of it when it's no longer needed.

As an example, the `IMessageWriter` interface defines the `Write` method:

C#

```
namespace DependencyInjection.Example;

public interface IMessageWriter
{
    void Write(string message);
}
```

This interface is implemented by a concrete type, `MessageWriter`:

C#

```
namespace DependencyInjection.Example;

public class MessageWriter : IMessageWriter
{
    public void Write(string message)
    {
        Console.WriteLine($"MessageWriter.Write(message: \"{mes-
```

```
sage}\")");
        }
}
```

The sample code registers the `IMessageWriter` service with the concrete type `MessageWriter`. The AddSingleton method registers the service with a singleton lifetime, the lifetime of a single request. Service lifetimes are described later in this article.

C#

```
using DependencyInjection.Example;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddHostedService<Worker>();
builder.Services.AddSingleton<IMessageWriter, MessageWriter>();

using IHost host = builder.Build();

host.Run();
```

In the preceding code, the sample app:

- Creates a host app builder instance.
- Configures the services by registering:
  - The `Worker` as a hosted service. For more information, see Worker Services in .NET.
  - The `IMessageWriter` interface as a singleton service with a corresponding implementation of the `MessageWriter` class.
- Builds the host and runs it.

The host contains the dependency injection service provider. It also contains all the other relevant services required to automatically instantiate the `Worker` and provide the corresponding `IMessageWriter` implementation as an argument.

C#

```
namespace DependencyInjection.Example;

public sealed class Worker : BackgroundService
{
    private readonly IMessageWriter _messageWriter;

    public Worker(IMessageWriter messageWriter) =>
```

```
        _messageWriter = messageWriter;

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            _messageWriter.Write($"Worker running at:
{DateTimeOffset.Now}");
            await Task.Delay(1_000, stoppingToken);
        }
    }
}
```

By using the DI pattern, the worker service:

- Doesn't use the concrete type `MessageWriter`, only the `IMessageWriter` interface that implements it. That makes it easy to change the implementation that the worker service uses without modifying the worker service.
- Doesn't create an instance of `MessageWriter`. The instance is created by the DI container.

The implementation of the `IMessageWriter` interface can be improved by using the built-in logging API:

C#

```
namespace DependencyInjection.Example;

public class LoggingMessageWriter : IMessageWriter
{
    private readonly ILogger<LoggingMessageWriter> _logger;

    public LoggingMessageWriter(ILogger<LoggingMessageWriter> log-
ger) =>
        _logger = logger;

    public void Write(string message) =>
        _logger.LogInformation("Info: {Msg}", message);
}
```

The updated `AddSingleton` method registers the new `IMessageWriter` implementation:

C#

```
builder.Services.AddSingleton<IMessageWriter, LoggingMessageWriter>
());
```

The HostApplicationBuilder (`builder`) type is part of the `Microsoft.Extensions.Hosting` NuGet package.

`LoggingMessageWriter` depends on ILogger<TCategoryName>, which it requests in the constructor. `ILogger<TCategoryName>` is a framework-provided service.

It's not unusual to use dependency injection in a chained fashion. Each requested dependency in turn requests its own dependencies. The container resolves the dependencies in the graph and returns the fully resolved service. The collective set of dependencies that must be resolved is typically referred to as a *dependency tree*, *dependency graph*, or *object graph*.

The container resolves `ILogger<TCategoryName>` by taking advantage of (generic) open types, eliminating the need to register every (generic) constructed type.

With dependency injection terminology, a service:

- Is typically an object that provides a service to other objects, such as the `IMessageWriter` service.
- Is not related to a web service, although the service may use a web service.

The framework provides a robust logging system. The `IMessageWriter` implementations shown in the preceding examples were written to demonstrate basic DI, not to implement logging. Most apps shouldn't need to write loggers. The following code demonstrates using the default logging, which only requires the `Worker` to be registered as a hosted service AddHostedService:

C#

```
public class Worker : BackgroundService
{
    private readonly ILogger<Worker> _logger;

    public Worker(ILogger<Worker> logger) =>
        _logger = logger;

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            _logger.LogInformation("Worker running at: {time}",
DateTimeOffset.Now);
            await Task.Delay(1_000, stoppingToken);
        }
```

```
    }
}
```

Using the preceding code, there is no need to update `Program.cs`, because logging is provided by the framework.

Multiple constructor discovery rules

When a type defines more than one constructor, the service provider has logic for determining which constructor to use. The constructor with the most parameters where the types are DI-resolvable is selected. Consider the following C# example service:

C#

```
public class ExampleService
{
    public ExampleService()
    {
    }

    public ExampleService(ILogger<ExampleService> logger)
    {
        // omitted for brevity
    }

    public ExampleService(FooService fooService, BarService barSer-
vice)
    {
        // omitted for brevity
    }
}
```

In the preceding code, assume that logging has been added and is resolvable from the service provider but the `FooService` and `BarService` types are not. The constructor with the `ILogger<ExampleService>` parameter is used to resolve the `ExampleService` instance. Even though there's a constructor that defines more parameters, the `FooService` and `BarService` types are not DI-resolvable.

If there's ambiguity when discovering constructors, an exception is thrown. Consider the following C# example service:

C#

```
public class ExampleService
{
    public ExampleService()
```

```
    {
    }

    public ExampleService(ILogger<ExampleService> logger)
    {
        // omitted for brevity
    }

    public ExampleService(IOptions<ExampleOptions> options)
    {
        // omitted for brevity
    }
}
```

> ⚠ **Warning**
>
> The `ExampleService` code with ambiguous DI-resolvable type parameters would throw an exception. Do **not** do this—it's intended to show what is meant by "ambiguous DI-resolvable types".

In the preceding example, there are three constructors. The first constructor is parameterless and requires no services from the service provider. Assume that both logging and options have been added to the DI container and are DI-resolvable services. When the DI container attempts to resolve the `ExampleService` type, it will throw an exception, as the two constructors are ambiguous.

You can avoid ambiguity by defining a constructor that accepts both DI-resolvable types instead:

C#

```
public class ExampleService
{
    public ExampleService()
    {
    }

    public ExampleService(
        ILogger<ExampleService> logger,
        IOptions<ExampleOptions> options)
    {
        // omitted for brevity
    }
}
```

Register groups of services with extension methods

Microsoft Extensions uses a convention for registering a group of related services. The convention is to use a single `Add{GROUP_NAME}` extension method to register all of the services required by a framework feature. For example, the AddOptions extension method registers all of the services required for using options.

Framework-provided services

When using any of the available host or app builder patterns, defaults are applied and services are registered by the framework. Consider some of the most popular host and app builder patterns:

- Host.CreateDefaultBuilder()
- Host.CreateApplicationBuilder()
- WebHost.CreateDefaultBuilder()
- WebApplication.CreateBuilder()
- WebAssemblyHostBuilder.CreateDefault
- MauiApp.CreateBuilder

After creating a builder from any of these APIs, the `IServiceCollection` has services defined by the framework, depending on how the host was configured. For apps based on the .NET templates, the framework could register hundreds of services.

The following table lists a small sample of these framework-registered services:

Service Type	Lifetime
Microsoft.Extensions.DependencyInjection.IServiceScopeFactory	Singleton
IHostApplicationLifetime	Singleton
Microsoft.Extensions.Logging.ILogger<TCategoryName>	Singleton
Microsoft.Extensions.Logging.ILoggerFactory	Singleton
Microsoft.Extensions.ObjectPool.ObjectPoolProvider	Singleton
Microsoft.Extensions.Options.IConfigureOptions<TOptions>	Transient
Microsoft.Extensions.Options.IOptions<TOptions>	Singleton
System.Diagnostics.DiagnosticListener	Singleton

Service Type	Lifetime
System.Diagnostics.DiagnosticSource	Singleton

Service lifetimes

Services can be registered with one of the following lifetimes:

- Transient
- Scoped
- Singleton

The following sections describe each of the preceding lifetimes. Choose an appropriate lifetime for each registered service.

Transient

Transient lifetime services are created each time they're requested from the service container. This lifetime works best for lightweight, stateless services. Register transient services with AddTransient.

In apps that process requests, transient services are disposed at the end of the request.

Scoped

For web applications, a scoped lifetime indicates that services are created once per client request (connection). Register scoped services with AddScoped.

In apps that process requests, scoped services are disposed at the end of the request.

When using Entity Framework Core, the AddDbContext extension method registers `DbContext` types with a scoped lifetime by default.

> **Note**
>
> Do ***not*** resolve a scoped service from a singleton and be careful not to do so indirectly, for example, through a transient service. It may cause the service to have incorrect state when processing subsequent requests. It's fine to:
>
> - Resolve a singleton service from a scoped or transient service.
> - Resolve a scoped service from another scoped or transient service.

By default, in the development environment, resolving a service from another service with a longer lifetime throws an exception. For more information, see Scope validation.

Singleton

Singleton lifetime services are created either:

- The first time they're requested.
- By the developer, when providing an implementation instance directly to the container. This approach is rarely needed.

Every subsequent request of the service implementation from the dependency injection container uses the same instance. If the app requires singleton behavior, allow the service container to manage the service's lifetime. Don't implement the singleton design pattern and provide code to dispose of the singleton. Services should never be disposed by code that resolved the service from the container. If a type or factory is registered as a singleton, the container disposes the singleton automatically.

Register singleton services with AddSingleton. Singleton services must be thread safe and are often used in stateless services.

In apps that process requests, singleton services are disposed when the ServiceProvider is disposed on application shutdown. Because memory is not released until the app is shut down, consider memory use with a singleton service.

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:	Yes	Yes	Yes

Method	Automatic object disposal	Multiple implementations	Pass args
`services.AddSingleton<IMyDep>(sp => new MyDep());` `services.AddSingleton<IMyDep>(sp => new MyDep(99));`			

`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

For more information on type disposal, see the Disposal of services section.

Registering a service with only an implementation type is equivalent to registering that service with the same implementation and service type. This is why multiple implementations of a service cannot be registered using the methods that don't take an explicit service type. These methods can register multiple *instances* of a service, but they will all have the same *implementation* type.

Any of the above service registration methods can be used to register multiple service instances of the same service type. In the following example, `AddSingleton` is called twice with `IMessageWriter` as the service type. The second call to `AddSingleton` overrides the previous one when resolved as `IMessageWriter` and adds to the previous one when multiple services are resolved via `IEnumerable<IMessageWriter>`. Services appear in the order they were registered when resolved via `IEnumerable<{SERVICE}>`.

C#

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

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddSingleton<IMessageWriter, ConsoleMessageWriter>
();
builder.Services.AddSingleton<IMessageWriter, LoggingMessageWriter>
();
builder.Services.AddSingleton<ExampleService>();

using IHost host = builder.Build();

_ = host.Services.GetService<ExampleService>();

await host.RunAsync();
```

The preceding sample source code registers two implementations of the `IMessageWriter`.

C#

```
using System.Diagnostics;

namespace ConsoleDI.IEnumerableExample;

public sealed class ExampleService
{
    public ExampleService(
        IMessageWriter messageWriter,
        IEnumerable<IMessageWriter> messageWriters)
    {
        Trace.Assert(messageWriter is LoggingMessageWriter);

        var dependencyArray = messageWriters.ToArray();
        Trace.Assert(dependencyArray[0] is ConsoleMessageWriter);
        Trace.Assert(dependencyArray[1] is LoggingMessageWriter);
    }
}
```

The `ExampleService` defines two constructor parameters; a single `IMessageWriter`, and an `IEnumerable<IMessageWriter>`. The single `IMessageWriter` is the last implementation to have been registered, whereas the `IEnumerable<IMessageWriter>` represents all registered implementations.

The framework also provides `TryAdd{LIFETIME}` extension methods, which register the service only if there isn't already an implementation registered.

In the following example, the call to `AddSingleton` registers `ConsoleMessageWriter` as an implementation for `IMessageWriter`. The call to `TryAddSingleton` has no effect because `IMessageWriter` already has a registered implementation:

C#

```
services.AddSingleton<IMessageWriter, ConsoleMessageWriter>();
services.TryAddSingleton<IMessageWriter, LoggingMessageWriter>();
```

The `TryAddSingleton` has no effect, as it was already added and the "try" will fail. The `ExampleService` would assert the following:

C#

```
public class ExampleService
{
    public ExampleService(
        IMessageWriter messageWriter,
        IEnumerable<IMessageWriter> messageWriters)
    {
        Trace.Assert(messageWriter is ConsoleMessageWriter);
        Trace.Assert(messageWriters.Single() is
ConsoleMessageWriter);
    }
}
```

For more information, see:

- TryAdd
- TryAddTransient
- TryAddScoped
- TryAddSingleton

The TryAddEnumerable(ServiceDescriptor) methods register the service only if there isn't already an implementation *of the same type*. Multiple services are resolved via `IEnumerable<{SERVICE}>`. When registering services, add an instance if one of the same types hasn't already been added. Library authors use `TryAddEnumerable` to avoid registering multiple copies of an implementation in the container.

In the following example, the first call to `TryAddEnumerable` registers `MessageWriter` as an implementation for `IMessageWriter1`. The second call registers `MessageWriter`

for `IMessageWriter2`. The third call has no effect because `IMessageWriter1` already has a registered implementation of `MessageWriter`:

C#

```
public interface IMessageWriter1 { }
public interface IMessageWriter2 { }

public class MessageWriter : IMessageWriter1, IMessageWriter2
{
}

services.TryAddEnumerable(ServiceDescriptor.Singleton<IMessageWriter1
, MessageWriter>());
services.TryAddEnumerable(ServiceDescriptor.Singleton<IMessageWriter2
, MessageWriter>());
services.TryAddEnumerable(ServiceDescriptor.Singleton<IMessageWriter1
, MessageWriter>());
```

Service registration is generally order-independent except when registering multiple implementations of the same type.

`IServiceCollection` is a collection of ServiceDescriptor objects. The following example shows how to register a service by creating and adding a `ServiceDescriptor`:

C#

```
string secretKey = Configuration["SecretKey"];
var descriptor = new ServiceDescriptor(
    typeof(IMessageWriter),
    _ => new DefaultMessageWriter(secretKey),
    ServiceLifetime.Transient);

services.Add(descriptor);
```

The built-in `Add{LIFETIME}` methods use the same approach. For example, see the AddScoped source code .

Constructor injection behavior

Services can be resolved by using:

- IServiceProvider
- ActivatorUtilities:
  - Creates objects that aren't registered in the container.
  - Used with some framework features.

Constructors can accept arguments that aren't provided by dependency injection, but the arguments must assign default values.

When services are resolved by `IServiceProvider` or `ActivatorUtilities`, constructor injection requires a *public* constructor.

When services are resolved by `ActivatorUtilities`, constructor injection requires that only one applicable constructor exists. Constructor overloads are supported, but only one overload can exist whose arguments can all be fulfilled by dependency injection.

Scope validation

When the app runs in the `Development` environment and calls CreateApplicationBuilder to build the host, the default service provider performs checks to verify that:

- Scoped services aren't resolved from the root service provider.
- Scoped services aren't injected into singletons.

The root service provider is created when BuildServiceProvider is called. The root service provider's lifetime corresponds to the app's lifetime when the provider starts with the app and is disposed when the app shuts down.

Scoped services are disposed by the container that created them. If a scoped service is created in the root container, the service's lifetime is effectively promoted to singleton because it's only disposed by the root container when the app shuts down. Validating service scopes catches these situations when `BuildServiceProvider` is called.

Scope scenarios

The IServiceScopeFactory is always registered as a singleton, but the IServiceProvider can vary based on the lifetime of the containing class. For example, if you resolve services from a scope, and any of those services take an IServiceProvider, it'll be a scoped instance.

To achieve scoping services within implementations of IHostedService, such as the BackgroundService, do *not* inject the service dependencies via constructor injection. Instead, inject IServiceScopeFactory, create a scope, then resolve dependencies from the scope to use the appropriate service lifetime.

C#

```
namespace WorkerScope.Example;
```

```
public sealed class Worker : BackgroundService
{
    private readonly ILogger<Worker> _logger;
    private readonly IServiceScopeFactory _serviceScopeFactory;

    public Worker(ILogger<Worker> logger, IServiceScopeFactory ser-
viceScopeFactory) =>
        (_logger, _serviceScopeFactory) = (logger,
serviceScopeFactory);

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            using (IServiceScope scope =
_serviceScopeFactory.CreateScope())
            {
                try
                {
                    _logger.LogInformation(
                        "Starting scoped work, provider hash:
{hash}.",
                        scope.ServiceProvider.GetHashCode());

                    var store =
scope.ServiceProvider.GetRequiredService<IObjectStore>();
                    var next = await store.GetNextAsync();
                    _logger.LogInformation("{next}", next);

                    var processor =
scope.ServiceProvider.GetRequiredService<IObjectProcessor>();
                    await processor.ProcessAsync(next);
                    _logger.LogInformation("Processing {name}.",
next.Name);

                    var relay =
scope.ServiceProvider.GetRequiredService<IObjectRelay>();
                    await relay.RelayAsync(next);
                    _logger.LogInformation("Processed results have
been relayed.");

                    var marked = await store.MarkAsync(next);
                    _logger.LogInformation("Marked as processed:
{next}", marked);
                }
                finally
                {
                    _logger.LogInformation(
                        "Finished scoped work, provider hash: {hash}.
{nl}",
                        scope.ServiceProvider.GetHashCode(),
Environment.NewLine);
                }
            }
```

```
            }
        }
    }
```

In the preceding code, while the app is running, the background service:

- Depends on the IServiceScopeFactory.
- Creates an IServiceScope for resolving additional services.
- Resolves scoped services for consumption.
- Works on processing objects and then relaying them, and finally marks them as processed.

From the sample source code, you can see how implementations of IHostedService can benefit from scoped service lifetimes.

See also

- Use dependency injection in .NET
- Dependency injection guidelines
- Dependency injection in ASP.NET Core
- NDC Conference Patterns for DI app development
- Explicit dependencies principle
- Inversion of control containers and the dependency injection pattern (Martin Fowler)
- DI bugs should be created in the github.com/dotnet/extensions repo