Dependency injection in ASP.NET Core

Article • 11/06/2023

By Kirk Larkin, Steve Smith, and Brandon Dahler

ASP.NET Core supports the dependency injection (DI) software design pattern, which is a technique for achieving Inversion of Control (IoC) between classes and their dependencies.

For more information specific to dependency injection within MVC controllers, see Dependency injection into controllers in ASP.NET Core.

For information on using dependency injection in applications other than web apps, see Dependency injection in .NET.

For more information on dependency injection of options, see Options pattern in ASP.NET Core.

This topic provides information on dependency injection in ASP.NET Core. The primary documentation on using dependency injection is contained in Dependency injection in .NET.

View or download sample code (how to download)

Overview of dependency injection

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

C#

```
public class MyDependency
{
    public void WriteMessage(string message)
    {
        Console.WriteLine($"MyDependency.WriteMessage called. Message: {message}");
    }
}
```

A class can create an instance of the `MyDependency` class to make use of its `WriteMessage` method. In the following example, the `MyDependency` class is a dependency of the `IndexModel` class:

C#

```
public class IndexModel : PageModel
{
    private readonly MyDependency _dependency = new MyDependency();

    public void OnGet()
    {
        _dependency.WriteMessage("IndexModel.OnGet");
    }
}
```

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

- To replace `MyDependency` with a different implementation, the `IndexModel` class must be modified.
- If `MyDependency` has dependencies, they must also be configured by the `IndexModel` class. In a large project with multiple classes depending on `MyDependency`, the configuration code becomes scattered across the app.
- This implementation is difficult to unit test.

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. ASP.NET Core provides a built-in service container, IServiceProvider. Services are typically registered in the app's `Program.cs` file.
- *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.

In the sample app, the `IMyDependency` interface defines the `WriteMessage` method:

C#

```
public interface IMyDependency
{
    void WriteMessage(string message);
}
```

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

C#

```
public class MyDependency : IMyDependency
{
    public void WriteMessage(string message)
    {
        Console.WriteLine($"MyDependency.WriteMessage Message: {message}");
    }
}
```

The sample app registers the `IMyDependency` service with the concrete type `MyDependency`. The AddScoped method registers the service with a scoped lifetime, the lifetime of a single request. Service lifetimes are described later in this topic.

C#

```
using DependencyInjectionSample.Interfaces;
using DependencyInjectionSample.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddScoped<IMyDependency, MyDependency>();

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

In the sample app, the `IMyDependency` service is requested and used to call the `WriteMessage` method:

C#

```
public class Index2Model : PageModel
{
    private readonly IMyDependency _myDependency;

    public Index2Model(IMyDependency myDependency)
    {
        _myDependency = myDependency;
    }

    public void OnGet()
    {
        _myDependency.WriteMessage("Index2Model.OnGet");
    }
}
```

By using the DI pattern, the controller or Razor Page:

- Doesn't use the concrete type `MyDependency`, only the `IMyDependency` interface it implements. That makes it easy to change the implementation without modifying the controller or Razor Page.
- Doesn't create an instance of `MyDependency`, it's created by the DI container.

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

C#

```
public class MyDependency2 : IMyDependency
{
    private readonly ILogger<MyDependency2> _logger;

    public MyDependency2(ILogger<MyDependency2> logger)
    {
        _logger = logger;
    }

    public void WriteMessage(string message)
```

```
    {
        _logger.LogInformation( $"MyDependency2.WriteMessage Message: {message}");
    }
}
```

The updated `Program.cs` registers the new `IMyDependency` implementation:

```
C#

using DependencyInjectionSample.Interfaces;
using DependencyInjectionSample.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddScoped<IMyDependency, MyDependency2>();

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

`MyDependency2` 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.

In dependency injection terminology, a service:

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

The framework provides a robust logging system. The `IMyDependency` 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 doesn't require any services to be registered:

```
C#

public class AboutModel : PageModel
{
    private readonly ILogger _logger;

    public AboutModel(ILogger<AboutModel> logger)
    {
        _logger = logger;
    }

    public string Message { get; set; } = string.Empty;

    public void OnGet()
    {
        Message = $"About page visited at {DateTime.UtcNow.ToLongTimeString()}";
        _logger.LogInformation(Message);
    }
}
```

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

Register groups of services with extension methods

The ASP.NET Core framework 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 AddControllers extension method registers the services required for MVC controllers.

The following code is generated by the Razor Pages template using individual user accounts and shows how to add additional services to the container using the extension methods AddDbContext and AddDefaultIdentity:

```
C#
```

```
using DependencyInjectionSample.Data;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();

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

Consider the following which registers services and configures options:

C#

```
using ConfigSample.Options;
using Microsoft.Extensions.DependencyInjection.ConfigSample.Options;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.Configure<PositionOptions>(
    builder.Configuration.GetSection(PositionOptions.Position));
builder.Services.Configure<ColorOptions>(
    builder.Configuration.GetSection(ColorOptions.Color));

builder.Services.AddScoped<IMyDependency, MyDependency>();
builder.Services.AddScoped<IMyDependency2, MyDependency2>();

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

Related groups of registrations can be moved to an extension method to register services. For example, the configuration services are added to the following class:

C#

```
using ConfigSample.Options;
using Microsoft.Extensions.Configuration;

namespace Microsoft.Extensions.DependencyInjection
{
    public static class MyConfigServiceCollectionExtensions
    {
        public static IServiceCollection AddConfig(
             this IServiceCollection services, IConfiguration config)
        {
            services.Configure<PositionOptions>(
                config.GetSection(PositionOptions.Position));
            services.Configure<ColorOptions>(
                config.GetSection(ColorOptions.Color));

            return services;
        }

        public static IServiceCollection AddMyDependencyGroup(
             this IServiceCollection services)
        {
            services.AddScoped<IMyDependency, MyDependency>();
            services.AddScoped<IMyDependency2, MyDependency2>();

            return services;
        }
    }
}
```

The remaining services are registered in a similar class. The following code uses the new extension methods to register the services:

C#

```
using Microsoft.Extensions.DependencyInjection.ConfigSample.Options;

var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddConfig(builder.Configuration)
    .AddMyDependencyGroup();

builder.Services.AddRazorPages();

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

Note: Each `services.Add{GROUP_NAME}` extension method adds and potentially configures services. For example, AddControllersWithViews adds the services MVC controllers with views require, and AddRazorPages adds the services Razor Pages requires.

Service lifetimes

See Service lifetimes in Dependency injection in .NET

To use scoped services in middleware, use one of the following approaches:

- Inject the service into the middleware's `Invoke` or `InvokeAsync` method. Using constructor injection throws a runtime exception because it forces the scoped service to behave like a singleton. The sample in the Lifetime and registration options section demonstrates the `InvokeAsync` approach.
- Use Factory-based middleware. Middleware registered using this approach is activated per client request (connection), which allows scoped services to be injected into the middleware's constructor.

For more information, see Write custom ASP.NET Core middleware.

Service registration methods

See Service registration methods in Dependency injection in .NET

It's common to use multiple implementations when mocking types for testing.

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 `IMyDependency` as the service type. The second call to `AddSingleton` overrides the previous one when resolved as `IMyDependency` and adds to the previous one when multiple services are resolved via `IEnumerable<IMyDependency>`. Services appear in the order they were registered when resolved via `IEnumerable<{SERVICE}>`.

C#

```
services.AddSingleton<IMyDependency, MyDependency>();
services.AddSingleton<IMyDependency, DifferentDependency>();

public class MyService
{
    public MyService(IMyDependency myDependency,
       IEnumerable<IMyDependency> myDependencies)
    {
        Trace.Assert(myDependency is DifferentDependency);

        var dependencyArray = myDependencies.ToArray();
        Trace.Assert(dependencyArray[0] is MyDependency);
        Trace.Assert(dependencyArray[1] is DifferentDependency);
    }
}
```

Keyed services

Keyed services refers to a mechanism for registering and retrieving Dependency Injection (DI) services using keys. A service is associated with a key by calling AddKeyedSingleton (or `AddKeyedScoped` or `AddKeyedTransient`) to register it. Access a registered service by specifying the key with the [FromKeyedServices] attribute. The following code shows how to use keyed services:

C#

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.SignalR;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddKeyedSingleton<ICache, BigCache>("big");
builder.Services.AddKeyedSingleton<ICache, SmallCache>("small");
builder.Services.AddControllers();

var app = builder.Build();

app.MapGet("/big", ([FromKeyedServices("big")] ICache bigCache) => bigCache.Get("date"));
app.MapGet("/small", ([FromKeyedServices("small")] ICache smallCache) =>
                                                        smallCache.Get("date"));

app.MapControllers();

app.Run();

public interface ICache
{
    object Get(string key);
}
public class BigCache : ICache
{
    public object Get(string key) => $"Resolving {key} from big cache.";
}

public class SmallCache : ICache
{
    public object Get(string key) => $"Resolving {key} from small cache.";
}

[ApiController]
[Route("/cache")]
public class CustomServicesApiController : Controller
{
    [HttpGet("big-cache")]
    public ActionResult<object> GetOk([FromKeyedServices("big")] ICache cache)
    {
        return cache.Get("data-mvc");
    }
}

public class MyHub : Hub
{
    public void Method([FromKeyedServices("small")] ICache cache)
    {
        Console.WriteLine(cache.Get("signalr"));
    }
}
```

Constructor injection behavior

See Constructor injection behavior in Dependency injection in .NET

Entity Framework contexts

By default, Entity Framework contexts are added to the service container using the scoped lifetime because web app database operations are normally scoped to the client request. To use a different lifetime, specify the lifetime by using an AddDbContext overload. Services of a given lifetime shouldn't use a database context with a lifetime that's shorter than the service's lifetime.

Lifetime and registration options

To demonstrate the difference between service lifetimes and their registration options, consider the following interfaces that represent a task as an operation with an identifier, `OperationId`. Depending on how the lifetime of an operation's service is configured for the following interfaces, the container provides either the same or different instances of the service when requested by a class:

C#

```
public interface IOperation
{
    string OperationId { get; }
}

public interface IOperationTransient : IOperation { }
public interface IOperationScoped : IOperation { }
public interface IOperationSingleton : IOperation { }
```

The following `Operation` class implements all of the preceding interfaces. The `Operation` constructor generates a GUID and stores the last 4 characters in the `OperationId` property:

C#

```
public class Operation : IOperationTransient, IOperationScoped, IOperationSingleton
{
    public Operation()
    {
        OperationId = Guid.NewGuid().ToString()[^4..];
    }

    public string OperationId { get; }
}
```

The following code creates multiple registrations of the `Operation` class according to the named lifetimes:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddTransient<IOperationTransient, Operation>();
builder.Services.AddScoped<IOperationScoped, Operation>();
builder.Services.AddSingleton<IOperationSingleton, Operation>();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseMyMiddleware();
app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The sample app demonstrates object lifetimes both within and between requests. The `IndexModel` and the middleware request each kind of `IOperation` type and log the `OperationId` for each:

C#

```
public class IndexModel : PageModel
{
    private readonly ILogger _logger;
    private readonly IOperationTransient _transientOperation;
    private readonly IOperationSingleton _singletonOperation;
    private readonly IOperationScoped _scopedOperation;

    public IndexModel(ILogger<IndexModel> logger,
                      IOperationTransient transientOperation,
                      IOperationScoped scopedOperation,
                      IOperationSingleton singletonOperation)
    {
        _logger = logger;
        _transientOperation = transientOperation;
        _scopedOperation    = scopedOperation;
```

```
        _singletonOperation = singletonOperation;
    }

    public void  OnGet()
    {
        _logger.LogInformation("Transient: " + _transientOperation.OperationId);
        _logger.LogInformation("Scoped: "    + _scopedOperation.OperationId);
        _logger.LogInformation("Singleton: " + _singletonOperation.OperationId);
    }
}
```

Similar to the `IndexModel`, the middleware resolves the same services:

C#

```
public class MyMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ILogger _logger;

    private readonly IOperationSingleton _singletonOperation;

    public MyMiddleware(RequestDelegate next, ILogger<MyMiddleware> logger,
        IOperationSingleton singletonOperation)
    {
        _logger = logger;
        _singletonOperation = singletonOperation;
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context,
        IOperationTransient transientOperation, IOperationScoped scopedOperation)
    {
        _logger.LogInformation("Transient: " + transientOperation.OperationId);
        _logger.LogInformation("Scoped: " + scopedOperation.OperationId);
        _logger.LogInformation("Singleton: " + _singletonOperation.OperationId);

        await _next(context);
    }
}

public static class MyMiddlewareExtensions
{
    public static IApplicationBuilder UseMyMiddleware(this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<MyMiddleware>();
    }
}
```

Scoped and transient services must be resolved in the `InvokeAsync` method:

C#

```
public async Task InvokeAsync(HttpContext context,
    IOperationTransient transientOperation, IOperationScoped scopedOperation)
{
    _logger.LogInformation("Transient: " + transientOperation.OperationId);
    _logger.LogInformation("Scoped: " + scopedOperation.OperationId);
    _logger.LogInformation("Singleton: " + _singletonOperation.OperationId);

    await _next(context);
}
```

The logger output shows:

- *Transient* objects are always different. The transient `OperationId` value is different in the `IndexModel` and in the middleware.
- *Scoped* objects are the same for a given request but differ across each new request.
- *Singleton* objects are the same for every request.

To reduce the logging output, set "Logging:LogLevel:Microsoft:Error" in the `appsettings.Development.json` file:

JSON

```
{
  "MyKey": "MyKey from appsettings.Developement.json",
  "Logging": {
```

```
    "LogLevel": {
      "Default": "Information",
      "System": "Debug",
      "Microsoft": "Error"
    }
  }
}
```

Resolve a service at app start up

The following code shows how to resolve a scoped service for a limited duration when the app starts:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<IMyDependency, MyDependency>();

var app = builder.Build();

using (var serviceScope = app.Services.CreateScope())
{
    var services = serviceScope.ServiceProvider;

    var myDependency = services.GetRequiredService<IMyDependency>();
    myDependency.WriteMessage("Call services from main");
}

app.MapGet("/", () => "Hello World!");

app.Run();
```

Scope validation

See Constructor injection behavior in Dependency injection in .NET

For more information, see Scope validation.

Request Services

Services and their dependencies within an ASP.NET Core request are exposed through HttpContext.RequestServices.

The framework creates a scope per request, and `RequestServices` exposes the scoped service provider. All scoped services are valid for as long as the request is active.

> ⓘ Note
>
> Prefer requesting dependencies as constructor parameters over resolving services from `RequestServices`. Requesting dependencies as constructor parameters yields classes that are easier to test.

Design services for dependency injection

When designing services for dependency injection:

- Avoid stateful, static classes and members. Avoid creating global state by designing apps to use singleton services instead.
- Avoid direct instantiation of dependent classes within services. Direct instantiation couples the code to a particular implementation.
- Make services small, well-factored, and easily tested.

If a class has a lot of injected dependencies, it might be a sign that the class has too many responsibilities and violates the Single Responsibility Principle (SRP). Attempt to refactor the class by moving some of its responsibilities into new classes. Keep in mind that Razor Pages page model classes and MVC controller classes should focus on UI concerns.

Disposal of services

The container calls Dispose for the IDisposable types it creates. Services resolved from the container should never be disposed by the developer. If a type or factory is registered as a singleton, the container disposes the singleton automatically.

In the following example, the services are created by the service container and disposed automatically: dependency-injection\samples\6.x\DIsample2\DIsample2\Services\Service1.cs

C#

```
public class Service1 : IDisposable
{
    private bool _disposed;

    public void Write(string message)
    {
        Console.WriteLine($"Service1: {message}");
    }

    public void Dispose()
    {
        if (_disposed)
            return;

        Console.WriteLine("Service1.Dispose");
        _disposed = true;
    }
}

public class Service2 : IDisposable
{
    private bool _disposed;

    public void Write(string message)
    {
        Console.WriteLine($"Service2: {message}");
    }

    public void Dispose()
    {
        if (_disposed)
            return;

        Console.WriteLine("Service2.Dispose");
        _disposed = true;
    }
}

public interface IService3
{
    public void Write(string message);
}

public class Service3 : IService3, IDisposable
{
    private bool _disposed;

    public Service3(string myKey)
    {
        MyKey = myKey;
    }

    public string MyKey { get; }

    public void Write(string message)
    {
        Console.WriteLine($"Service3: {message}, MyKey = {MyKey}");
    }

    public void Dispose()
    {
        if (_disposed)
            return;

        Console.WriteLine("Service3.Dispose");
        _disposed = true;
    }
}
```

C#

```
using DIsample2.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddScoped<Service1>();
builder.Services.AddSingleton<Service2>();

var myKey = builder.Configuration["MyKey"];
builder.Services.AddSingleton<IService3>(sp => new Service3(myKey));

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

C#

```
public class IndexModel : PageModel
{
    private readonly Service1 _service1;
    private readonly Service2 _service2;
    private readonly IService3 _service3;

    public IndexModel(Service1 service1, Service2 service2, IService3 service3)
    {
        _service1 = service1;
        _service2 = service2;
        _service3 = service3;
    }

    public void OnGet()
    {
        _service1.Write("IndexModel.OnGet");
        _service2.Write("IndexModel.OnGet");
        _service3.Write("IndexModel.OnGet");
    }
}
```

The debug console shows the following output after each refresh of the Index page:

Console

```
Service1: IndexModel.OnGet
Service2: IndexModel.OnGet
Service3: IndexModel.OnGet, MyKey = MyKey from appsettings.Developement.json
Service1.Dispose
```

Services not created by the service container

Consider the following code:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddSingleton(new Service1());
builder.Services.AddSingleton(new Service2());
```

In the preceding code:

- The service instances aren't created by the service container.
- The framework doesn't dispose of the services automatically.
- The developer is responsible for disposing the services.

IDisposable guidance for Transient and shared instances

See IDisposable guidance for Transient and shared instance in Dependency injection in .NET

Default service container replacement

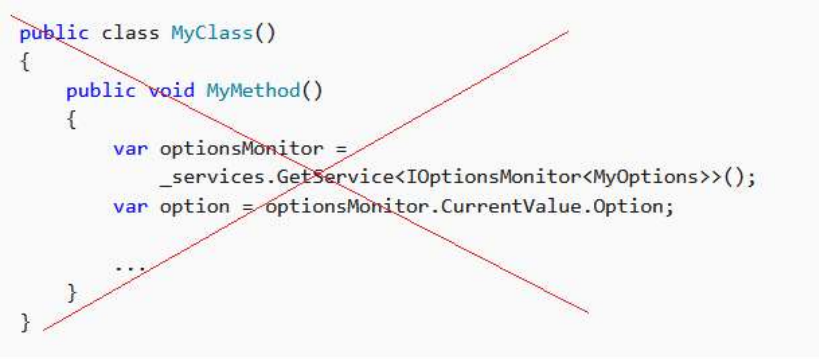
See Default service container replacement in Dependency injection in .NET

Recommendations

See Recommendations in Dependency injection in .NET

- Avoid using the *service locator pattern*. For example, don't invoke GetService to obtain a service instance when you can use DI instead:

  Incorrect:



```
public class MyClass()
{
    public void MyMethod()
    {
        var optionsMonitor =
            _services.GetService<IOptionsMonitor<MyOptions>>();
        var option = optionsMonitor.CurrentValue.Option;

        ...
    }
}
```

  Correct:

```C#
public class MyClass
{
    private readonly IOptionsMonitor<MyOptions> _optionsMonitor;

    public MyClass(IOptionsMonitor<MyOptions> optionsMonitor)
    {
        _optionsMonitor = optionsMonitor;
    }

    public void MyMethod()
    {
        var option = _optionsMonitor.CurrentValue.Option;

        ...
    }
}
```

- Another service locator variation to avoid is injecting a factory that resolves dependencies at runtime. Both of these practices mix Inversion of Control strategies.
- Avoid static access to `HttpContext` (for example, IHttpContextAccessor.HttpContext).

DI is an *alternative* to static/global object access patterns. You may not be able to realize the benefits of DI if you mix it with static object access.

Recommended patterns for multi-tenancy in DI

Orchard Core is an application framework for building modular, multi-tenant applications on ASP.NET Core. For more information, see the Orchard Core Documentation.

See the Orchard Core samples for examples of how to build modular and multi-tenant apps using just the Orchard Core Framework without any of its CMS-specific features.

Framework-provided services

`Program.cs` registers services that the app uses, including platform features, such as Entity Framework Core and ASP.NET Core MVC. Initially, the `IServiceCollection` provided to `Program.cs` has services defined by the framework depending on how the host was configured. For apps based on the ASP.NET Core templates, the framework registers more than 250 services.

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

Expand table

Service Type	Lifetime
Microsoft.AspNetCore.Hosting.Builder.IApplicationBuilderFactory	Transient
IHostApplicationLifetime	Singleton
IWebHostEnvironment	Singleton
Microsoft.AspNetCore.Hosting.IStartup	Singleton
Microsoft.AspNetCore.Hosting.IStartupFilter	Transient
Microsoft.AspNetCore.Hosting.Server.IServer	Singleton
Microsoft.AspNetCore.Http.IHttpContextFactory	Transient
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.DiagnosticSource	Singleton
System.Diagnostics.DiagnosticListener	Singleton

Additional resources

- Dependency injection into views in ASP.NET Core
- Dependency injection into controllers in ASP.NET Core
- Dependency injection in requirement handlers in ASP.NET Core
- ASP.NET Core Blazor dependency injection
- NDC Conference Patterns for DI app development
- App startup in ASP.NET Core
- Factory-based middleware activation in ASP.NET Core
- ASP.NET CORE DEPENDENCY INJECTION: WHAT IS THE ISERVICECOLLECTION?
- Four ways to dispose IDisposables in ASP.NET Core
- Writing Clean Code in ASP.NET Core with Dependency Injection (MSDN)
- Explicit Dependencies Principle
- Inversion of Control Containers and the Dependency Injection Pattern (Martin Fowler)
- How to register a service with multiple interfaces in ASP.NET Core DI