

# Dependency injection in ASP.NET Core

07/21/2020 • 40 minutes to read •      +17

## In this article

- [Overview of dependency injection](#)
- [Services injected into Startup](#)
- [Register groups of services with extension methods](#)
- [Service lifetimes](#)
- [Service registration methods](#)
- [Entity Framework contexts](#)
- [Lifetime and registration options](#)
- [Call services from main](#)
- [Scope validation](#)
- [Request Services](#)
- [Design services for dependency injection](#)
- [Default service container replacement](#)
- [Thread safety](#)
- [Recommendations](#)
- [Recommended patterns for multi-tenancy in DI](#)
- [Framework-provided services](#)
- [Additional resources](#)

By [Kirk Larkin](#), [Steve Smith](#), [Scott Addie](#), 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 more information on dependency injection of options, see [Options pattern in ASP.NET Core](#).

[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#

 Copy

```
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#

 Copy

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

    public void OnGet()
    {
```

```
        _dependency.WriteMessage("IndexModel.OnGet created this message.");  
    }  
}
```

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. The app should use a mock or stub `MyDependency` 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. ASP.NET Core provides a built-in service container, [IServiceProvider](#). Services are typically registered in the app's `Startup.ConfigureServices` method.
- *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#

 Copy

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

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

C#

 Copy

```
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#

 Copy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<IMyDependency, MyDependency>();

    services.AddRazorPages();
}
```

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

C#

 Copy

```
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:

- Doesn't use the concrete type `MyDependency`, only the `IMyDependency` interface it implements. That makes it easy to change the implementation that the controller uses without modifying the controller.
- 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#

 Copy

```
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 `ConfigureServices` method registers the new `IMyDependency` implementation:

C#

 Copy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<IMyDependency, MyDependency2>();

    services.AddRazorPages();
}
```

`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 in `ConfigureServices`:

C#



```
public class AboutModel : PageModel
{
    private readonly ILogger _logger;

    public AboutModel(ILogger<AboutModel> logger)
```

```
{
    _logger = logger;
}

public string Message { get; set; }

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

Using the preceding code, there is no need to update `ConfigureServices`, because [logging](#) is provided by the framework.

## Services injected into Startup

Services can be injected into the `Startup` constructor and the `Startup.Configure` method.

Only the following services can be injected into the `Startup` constructor when using the Generic Host ([IHostBuilder](#)):

- [IWebHostEnvironment](#)
- [IHostEnvironment](#)
- [IConfiguration](#)

Any service registered with the DI container can be injected into the `Startup.Configure` method:

C#

 Copy

```
public void Configure(IApplicationBuilder app, ILogger<Startup> logger)
{
    ...
}
```

For more information, see [App startup in ASP.NET Core](#) and [Access configuration in Startup](#).

## 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 `<Microsoft.Extensions.DependencyInjection.MvcServiceCollectionExtensions.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#



```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();
    services.AddRazorPages();
}
```

Consider the following `ConfigureServices` method, which registers services and configures options:

C#



```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<PositionOptions>(
        Configuration.GetSection(PositionOptions.Position));
    services.Configure<ColorOptions>(
```



```
Configuration.GetSection(ColorOptions.Color));

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

services.AddRazorPages();
}
```

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#

 Copy

```
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;
        }
    }
}
```

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

C#

 Copy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddConfig(Configuration)
        .AddMyDependencyGroup();

    services.AddRazorPages();
}
```

**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. We recommended that apps follow this naming convention. Place extension methods in the [Microsoft.Extensions.DependencyInjection](#) namespace to encapsulate groups of service registrations.

## 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

Scoped lifetime 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.

Do **not** resolve a scoped service from a singleton. 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](#).

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 `InvokeAsync` method.

For more information, see [Write custom ASP.NET Core middleware](#).

# 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 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.

### Warning

Do **not** resolve a scoped service from a singleton. 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.

## Service registration methods

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

Method	Automatic object disposal	Multiple implementations	Pass args
<code>Add{LIFETIME}&lt;{SERVICE}, {IMPLEMENTATION}&gt;()</code> Example: <code>services.AddSingleton&lt;IMyDep, MyDep&gt;();</code>	Yes	Yes	No


Method	Automatic object disposal	Multiple implementations	Pass args
<code>Add{LIFETIME}&lt;{SERVICE}&gt;(sp =&gt; new {IMPLEMENTATION})</code> Examples: <code>services.AddSingleton&lt;IMyDep&gt;(sp =&gt; new MyDep());</code> <code>services.AddSingleton&lt;IMyDep&gt;(sp =&gt; new MyDep(99));</code>	Yes	Yes	Yes
<code>Add{LIFETIME}&lt;{IMPLEMENTATION}&gt;()</code> Example: <code>services.AddSingleton&lt;MyDep&gt;();</code>	Yes	No	No
<code>AddSingleton&lt;{SERVICE}&gt;(new {IMPLEMENTATION})</code> Examples: <code>services.AddSingleton&lt;IMyDep&gt;(new MyDep());</code> <code>services.AddSingleton&lt;IMyDep&gt;(new MyDep(99));</code>	No	Yes	Yes
<code>AddSingleton(new {IMPLEMENTATION})</code> Examples: <code>services.AddSingleton(new MyDep());</code> <code>services.AddSingleton(new MyDep(99));</code>	No	No	Yes

For more information on type disposal, see the [Disposal of services](#) section. It's common to use multiple implementations when [mocking types for testing](#).

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 `MyDependency` as an implementation for `IMyDependency`. The call to `TryAddSingleton` has no effect because `IMyDependency` already has a registered implementation:

C#

 Copy

```
services.AddSingleton<IMyDependency, MyDependency>();  
// The following line has no effect:  
services.TryAddSingleton<IMyDependency, DifferentDependency>();
```

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, the developer should add an instance if one of the same type hasn't already been added. Generally, 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 `MyDependency` as an implementation for `IMyDependency1`. The second call registers `MyDependency` for `IMyDependency2`. The third call has no effect because `IMyDependency1` already has a registered implementation of `MyDependency`:

C#

 Copy

```
public interface IMyDependency1 { }  
public interface IMyDependency2 { }  
  
public class MyDependency : IMyDependency1, IMyDependency2 { }  
  
services.TryAddEnumerable(ServiceDescriptor.Singleton<IMyDependency1, MyDependency>());  
services.TryAddEnumerable(ServiceDescriptor.Singleton<IMyDependency2, MyDependency>());  
services.TryAddEnumerable(ServiceDescriptor.Singleton<IMyDependency1, MyDependency>());
```

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#



```
var myKey = Configuration["MyKey"];
var descriptor = new ServiceDescriptor(
    typeof(IMyDependency),
    sp => new MyDependency5(myKey),
    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 framework features, such as [Tag Helpers](#), MVC controllers, and [model binders](#).

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.

## 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#

 Copy

```
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 `Startup.ConfigureServices` method creates multiple registrations of the `Operation` class according to the named lifetimes:

C#



```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IOperationTransient, Operation>();
    services.AddScoped<IOperationScoped, Operation>();
    services.AddSingleton<IOperationSingleton, Operation>();

    services.AddRazorPages();
}
```

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#

 Copy

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

    private readonly IOperationTransient _transientOperation;
    private readonly IOperationSingleton _singletonOperation;
```

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

public async Task InvokeAsync(HttpContext context,
    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 services must be resolved in the `InvokeAsync` method:

C#

 Copy

```
public async Task InvokeAsync(HttpContext context,
    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 each request but different across each 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

 Copy

```
{  
  "MyKey": "MyKey from appsettings.Development.json",  
  "Logging": {  
    "LogLevel": {  
      "Default": "Information",  
      "System": "Debug",  
      "Microsoft": "Error"  
    }  
  }  
}
```

## Call services from main

Create an `IServiceScope` with `IServiceScopeFactory.CreateScope` to resolve a scoped service within the app's scope. This approach is useful to access a scoped service at startup to run initialization tasks.

The following example shows how to access the scoped `IMyDependency` service and call its `WriteMessage` method in `Program.Main`:

C#

 Copy

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = CreateHostBuilder(args).Build();

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

            try
            {
                var myDependency = services.GetRequiredService<IMyDependency>();
                myDependency.WriteMessage("Call services from main");
            }
            catch (Exception ex)
            {
                var logger = services.GetRequiredService<ILogger<Program>>();
                logger.LogError(ex, "An error occurred.");
            }
        }

        host.Run();
    }

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

# Scope validation

When the app runs in the [Development environment](#) and calls [CreateDefaultBuilder](#) 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.

For more information, see [Scope validation](#).

## Request Services

The services available within an ASP.NET Core request are exposed through the [HttpContext.RequestServices](#) collection. When services are requested from inside of a request, the services and their dependencies are resolved from the [RequestServices](#) collection.

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 to resolving services from the `RequestServices` collection. This results in 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:

C#

 Copy

```
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#

 Copy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<Service1>();
    services.AddSingleton<Service2>();

    var myKey = Configuration["MyKey"];
    services.AddSingleton<IService3>(sp => new Service3(myKey));
}
```

```
services.AddRazorPages();  
}
```

C#

 Copy

```
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

 Copy

```
Service1: IndexModel.OnGet  
Service2: IndexModel.OnGet  
Service3: IndexModel.OnGet  
Service1.Dispose
```

# Services not created by the service container

Consider the following code:

C#

 Copy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton(new Service1());
    services.AddSingleton(new Service2());

    services.AddRazorPages();
}
```

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

### Transient, limited lifetime

#### Scenario

The app requires an `IDisposable` instance with a transient lifetime for either of the following scenarios:

- The instance is resolved in the root scope (root container).
- The instance should be disposed before the scope ends.

## Solution

Use the factory pattern to create an instance outside of the parent scope. In this situation, the app would generally have a `Create` method that calls the final type's constructor directly. If the final type has other dependencies, the factory can:

- Receive an `IServiceProvider` in its constructor.
- Use `ActivatorUtilities.CreateInstance` to instantiate the instance outside of the container, while using the container for its dependencies.

## Shared instance, limited lifetime

### Scenario

The app requires a shared `IDisposable` instance across multiple services, but the `IDisposable` instance should have a limited lifetime.

### Solution

Register the instance with a scoped lifetime. Use `IServiceScopeFactory.CreateScope` to create a new `IServiceScope`. Use the scope's `IServiceProvider` to get required services. Dispose the scope when it's no longer needed.

## General IDisposable guidelines

- Don't register `IDisposable` instances with a transient lifetime. Use the factory pattern instead.
- Don't resolve `IDisposable` instances with a transient or scoped lifetime in the root scope. The only exception to this is if the app creates/recreates and disposes `IServiceProvider`, but this isn't an ideal pattern.
- Receiving an `IDisposable` dependency via DI doesn't require that the receiver implement `IDisposable` itself. The receiver of the `IDisposable` dependency shouldn't call `Dispose` on that dependency.
- Use scopes to control the lifetimes of services. Scopes aren't hierarchical, and there's no special connection among scopes.

# Default service container replacement

The built-in service container is designed to serve the needs of the framework and most consumer apps. We recommend using the built-in container unless you need a specific feature that it doesn't support, such as:

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

The following third-party containers can be used with ASP.NET Core apps:

- [Autofac](#)
- [Dryloc](#)
- [Grace](#)
- [LightInject](#)
- [Lamar](#)
- [Stashbox](#)
- [Unity](#)

## Thread safety

Create thread-safe singleton services. If a singleton service has a dependency on a transient service, the transient service may also require thread safety depending on how it's used by the singleton.

The factory method of single service, such as the second argument to [AddSingleton<TService>\(IServiceCollection, Func<IServiceProvider,TService>\)](#), doesn't need to be thread-safe. Like a type (`static`) constructor, it's guaranteed to be called only once by a single thread.


# Recommendations

- `async/await` and `Task` based service resolution isn't supported. Because C# doesn't support asynchronous constructors, use asynchronous methods after synchronously resolving the service.
- Avoid storing data and configuration directly in the service container. For example, a user's shopping cart shouldn't typically be added to the service container. Configuration should use the [options pattern](#). Similarly, avoid "data holder" objects that only exist to allow access to another object. It's better to request the actual item via DI.
- Avoid static access to services. For example, avoid capturing [IApplicationBuilder.ApplicationServices](#) as a static field or property for use elsewhere.
- Keep DI factories fast and synchronous.
- 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#  Copy

```
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](#)).
- Avoid calls to [BuildServiceProvider](#) in `ConfigureServices`. Calling `BuildServiceProvider` typically happens when the developer wants to resolve a service in `ConfigureServices`. For example, consider the case where the `LoginPath` is loaded from configuration. Avoid the following approach:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IMyService, MyService>();
    using (var serviceProvider = services.BuildServiceProvider())
    {
        var myService = serviceProvider.GetRequiredService<IMyService>();
        services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
            .AddCookie(options =>
            {
                options.LoginPath = myService.GetLoginPath();
            });
    };
    services.AddRazorPages();
}
```


In the preceding image, selecting the green wavy line under `services.BuildServiceProvider` shows the following ASP0000 warning:

ASP0000 Calling 'BuildServiceProvider' from application code results in an additional copy of singleton services being created. Consider alternatives such as dependency injecting services as parameters to 'Configure'.

Calling `BuildServiceProvider` creates a second container, which can create torn singletons and cause references to object graphs across multiple containers.

A correct way to get `LoginPath` is to use the options pattern's built-in support for DI:

C#

 Copy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
        .AddCookie();

    services.AddOptions<CookieAuthenticationOptions>(
        CookieAuthenticationDefaults.AuthenticationScheme)
```



```
.Configure<IMyService>((options, myService) =>
{
    options.LoginPath = myService.GetLoginPath();
});

services.AddRazorPages();
}
```

- Disposable transient services are captured by the container for disposal. This can turn into a memory leak if resolved from the top level container.
- Enable scope validation to make sure the app doesn't have singletons that capture scoped services. For more information, see [Scope validation](#).

Like all sets of recommendations, you may encounter situations where ignoring a recommendation is required. Exceptions are rare, mostly special cases within the framework itself.

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

The `Startup.ConfigureServices` method registers services that the app uses, including platform features, such as Entity Framework Core and ASP.NET Core MVC. Initially, the `IServiceCollection` provided to `ConfigureServices` 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:

Service Type	Lifetime
<a href="#">Microsoft.AspNetCore.Hosting.Builder.IApplicationBuilderFactory</a>	Transient
<a href="#">IHostApplicationLifetime</a>	Singleton
<a href="#">IWebHostEnvironment</a>	Singleton
<a href="#">Microsoft.AspNetCore.Hosting.IStartup</a>	Singleton
<a href="#">Microsoft.AspNetCore.Hosting.IStartupFilter</a>	Transient
<a href="#">Microsoft.AspNetCore.Hosting.Server.IServer</a>	Singleton
<a href="#">Microsoft.AspNetCore.Http.IHttpContextFactory</a>	Transient
<a href="#">Microsoft.Extensions.Logging.ILogger&lt;TCategoryName&gt;</a>	Singleton
<a href="#">Microsoft.Extensions.Logging.ILoggerFactory</a>	Singleton
<a href="#">Microsoft.Extensions.ObjectPool.ObjectPoolProvider</a>	Singleton
<a href="#">Microsoft.Extensions.Options.IConfigureOptions&lt;TOptions&gt;</a>	Transient

Service Type	Lifetime
<a href="#">Microsoft.Extensions.Options.IOptions&lt;TOptions&gt;</a>	Singleton
<a href="#">System.Diagnostics.DiagnosticSource</a>	Singleton
<a href="#">System.Diagnostics.DiagnosticListener</a>	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](#)
- [Four ways to dispose IDisposableables 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](#)

Is this page helpful?

 Yes  No