Dependency injection in ASP.NET Core

11/05/2019 • 19 minutes to read • +12

In this article

By Steve Smith, Scott Addie, and Luke Latham

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.

View or download sample code (how to download)

Overview of dependency injection

A *dependency* is any object that another object requires. Examine the following `MyDependency` class with a `WriteMessage` method that other classes in an app depend upon:

C# Copy

```csharp
public class MyDependency
{
    public MyDependency()
    {
    }

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

        return Task.FromResult(0);
    }
}
```

An instance of the `MyDependency` class can be created to make the `WriteMessage` method available to a class. The `MyDependency` class is a dependency of the `IndexModel` class:

C# Copy

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

    public async Task OnGetAsync()
    {
        await _dependency.WriteMessage(
            "IndexModel.OnGetAsync created this message.");
    }
}
```

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

- To replace `MyDependency` with a different implementation, the class must be modified.
- If `MyDependency` has dependencies, they must be configured by the 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 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 a method that the service provides to the app:

C# Copy

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

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

C# Copy

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

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

    public Task WriteMessage(string message)
    {
        _logger.LogInformation(
            "MyDependency.WriteMessage called. Message: {MESSAGE}",
            message);

        return Task.FromResult(0);
    }
}
```

`MyDependency` requests an ILogger<TCategoryName> in its constructor. 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*.

`IMyDependency` and `ILogger<TCategoryName>` must be registered in the service container. `IMyDependency` is registered in `Startup.ConfigureServices`. `ILogger<TCategoryName>` is registered by the logging abstractions infrastructure, so it's a framework-provided service registered by default by the framework.

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

C# Copy

```csharp
services.AddSingleton(typeof(ILogger<>), typeof(Logger<>));
```

In the sample app, the `IMyDependency` service is registered with the concrete type `MyDependency`. The registration scopes the service lifetime to the lifetime of a single request. Service lifetimes are described later in this topic.

C# Copy

```csharp
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    services.AddScoped<IMyDependency, MyDependency>();
    services.AddTransient<IOperationTransient, Operation>();
    services.AddScoped<IOperationScoped, Operation>();
    services.AddSingleton<IOperationSingleton, Operation>();
    services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));

    // OperationService depends on each of the other Operation types.
    services.AddTransient<OperationService, OperationService>();
}
```

Note

Each `services.Add{SERVICE_NAME}` extension method adds (and potentially configures) services. For example, `services.AddMvc()` adds the services Razor Pages and MVC require. We recommended that apps follow this convention. Place extension methods in the Microsoft.Extensions.DependencyInjection namespace to encapsulate groups of service registrations.

If the service's constructor requires a built in type, such as a `string`, the type can be injected by using configuration or the options pattern:

C# Copy

```csharp
public class MyDependency : IMyDependency
{
    public MyDependency(IConfiguration config)
    {
        var myStringValue = config["MyStringKey"];

        // Use myStringValue
    }

    ...
}
```

An instance of the service is requested via the constructor of a class where the service is used and assigned to a private field. The field is used to access the service as necessary throughout the class.

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

C# Copy

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

    public IndexModel(
        IMyDependency myDependency,
        OperationService operationService,
        IOperationTransient transientOperation,
        IOperationScoped scopedOperation,
        IOperationSingleton singletonOperation,
        IOperationSingletonInstance singletonInstanceOperation)
    {
        _myDependency = myDependency;
        OperationService = operationService;
        TransientOperation = transientOperation;
        ScopedOperation = scopedOperation;
        SingletonOperation = singletonOperation;
        SingletonInstanceOperation = singletonInstanceOperation;
    }

    public OperationService OperationService { get; }
    public IOperationTransient TransientOperation { get; }
    public IOperationScoped ScopedOperation { get; }
    public IOperationSingleton SingletonOperation { get; }
    public IOperationSingletonInstance SingletonInstanceOperation { get; }

    public async Task OnGetAsync()
    {
        await _myDependency.WriteMessage(
            "IndexModel.OnGetAsync created this message.");
    }
}
```

Services injected into Startup

Only the following service types can be injected into the `Startup` constructor when using the Generic Host (IHostBuilder):

- `IWebHostEnvironment`
- IHostEnvironment
- IConfiguration

Services can be injected into `Startup.Configure`:

C#

```
public void Configure(IApplicationBuilder app, IOptions<MyOptions> options)
{
    ...
}
```

For more information, see App startup in ASP.NET Core.

Framework-provided services

The `Startup.ConfigureServices` method is responsible for defining the 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. It's not uncommon for an app based on an ASP.NET Core template to have hundreds of services registered by the framework. A small sample of framework-registered services is listed in the following table.

Service Type	**Lifetime**
Microsoft.AspNetCore.Hosting.Builder.IApplicationBuilderFactory	Transient

Service Type	Lifetime
`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

Register additional services with extension methods

When a service collection extension method is available to register a service (and its dependent services, if required), the convention is to use a single `Add{SERVICE_NAME}` extension method to register all of the services required by that service. The following code is an example of how to add additional services to the container using the extension methods AddDbContext<TContext> and AddIdentityCore:

C# Copy

```csharp
public void ConfigureServices(IServiceCollection services)
{
    ...

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    ...
}
```

For more information, see the ServiceCollection class in the API documentation.

Service lifetimes

Choose an appropriate lifetime for each registered service. ASP.NET Core services can be configured with the following lifetimes:

Transient

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

Scoped

Scoped lifetime services (AddScoped) are created once per client request (connection).

> ⚠ **Warning**
>
> When using a scoped service in a middleware, inject the service into the `Invoke` or `InvokeAsync` method. Don't inject via constructor injection because it forces the service to behave like a singleton. For more information, see **Write custom ASP.NET Core middleware**.

Singleton

Singleton lifetime services (AddSingleton) are created the first time they're requested (or when `Startup.ConfigureServices` is run and an instance is specified with the service registration). Every subsequent request uses the same instance. If the app requires singleton behavior, allowing the service container to manage the service's lifetime is recommended. Don't implement the singleton design pattern and provide user code to manage the object's lifetime in the class.

> ⚠ **Warning**
>
> It's dangerous to resolve a scoped service from a singleton. It may cause the service to have incorrect state when processing subsequent requests.

Service registration methods

Service registration extension methods offer overloads that are useful in specific scenarios.

Method	Automatic object disposal	Multiple implementations	Pass args

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

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

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

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

`AddSingleton(new {IMPLEMENTATION})`
Examples:
`services.AddSingleton(new MyDep());`
`services.AddSingleton(new MyDep("A string!"));`
	No	No	Yes

For more information on type disposal, see the Disposal of services section. A common scenario for multiple implementations is mocking types for testing.

`TryAdd{LIFETIME}` methods only register the service if there isn't already an implementation registered.

In the following example, the first line registers `MyDependency` for `IMyDependency`. The second line 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

TryAddEnumerable(ServiceDescriptor) methods only register the service if there isn't already an implementation *of the same type*. Multiple services are resolved via `IEnumerable<{SERVICE}>`. When registering services, the developer only wants to add an instance if one of the same type hasn't already been added. Generally, this method is used by library authors to avoid registering two copies of an instance in the container.

In the following example, the first line registers `MyDep` for `IMyDep1`. The second line registers `MyDep` for `IMyDep2`. The third line has no effect because `IMyDep1` already has a registered implementation of `MyDep`:

C# Copy

```
public interface IMyDep1 {}
public interface IMyDep2 {}

public class MyDep : IMyDep1, IMyDep2 {}

services.TryAddEnumerable(ServiceDescriptor.Singleton<IMyDep1, MyDep>());
services.TryAddEnumerable(ServiceDescriptor.Singleton<IMyDep2, MyDep>());
// Two registrations of MyDep for IMyDep1 is avoided by the following line:
services.TryAddEnumerable(ServiceDescriptor.Singleton<IMyDep1, MyDep>());
```

Constructor injection behavior

Services can be resolved by two mechanisms:

- IServiceProvider
- ActivatorUtilities – Permits object creation without service registration in the dependency injection container. `ActivatorUtilities` is used with user-facing abstractions, 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

Entity Framework contexts are usually added to the service container using the scoped lifetime because web app database operations are normally scoped to the client request. The default lifetime is scoped if a lifetime isn't specified by an AddDbContext<TContext> overload when registering the database context. Services of a given lifetime shouldn't use a database context with a shorter lifetime than the service.

Lifetime and registration options

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

C# Copy

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

public interface IOperationTransient : IOperation
{
}

public interface IOperationScoped : IOperation
{
}

public interface IOperationSingleton : IOperation
{
}

public interface IOperationSingletonInstance : IOperation
{
}
```

The interfaces are implemented in the `Operation` class. The `Operation` constructor generates a GUID if one isn't supplied:

C# Copy

```
public class Operation : IOperationTransient,
    IOperationScoped,
    IOperationSingleton,
    IOperationSingletonInstance
{
    public Operation() : this(Guid.NewGuid())
    {
    }

    public Operation(Guid id)
    {
        OperationId = id;
    }

    public Guid OperationId { get; private set; }
}
```

An `OperationService` is registered that depends on each of the other `Operation` types. When `OperationService` is requested via dependency injection, it receives either a new instance of each service or an existing instance based on the lifetime of the dependent service.

- When transient services are created when requested from the container, the `OperationId` of the `IOperationTransient` service is different than the `OperationId` of the `OperationService`. `OperationService` receives a new instance of the `IOperationTransient` class. The new instance yields a different `OperationId`.
- When scoped services are created per client request, the `OperationId` of the `IOperationScoped` service is the same as that of `OperationService` within a client request. Across client requests, both services share a different `OperationId` value.
- When singleton and singleton-instance services are created once and used across all client requests and all services, the `OperationId` is constant across all service requests.

C# Copy

```csharp
public class OperationService
{
    public OperationService(
        IOperationTransient transientOperation,
        IOperationScoped scopedOperation,
        IOperationSingleton singletonOperation,
        IOperationSingletonInstance instanceOperation)
    {
        TransientOperation = transientOperation;
        ScopedOperation = scopedOperation;
        SingletonOperation = singletonOperation;
        SingletonInstanceOperation = instanceOperation;
    }

    public IOperationTransient TransientOperation { get; }
    public IOperationScoped ScopedOperation { get; }
    public IOperationSingleton SingletonOperation { get; }
    public IOperationSingletonInstance SingletonInstanceOperation { get; }
}
```

In `Startup.ConfigureServices`, each type is added to the container according to its named lifetime:

C# Copy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    services.AddScoped<IMyDependency, MyDependency>();
    services.AddTransient<IOperationTransient, Operation>();
    services.AddScoped<IOperationScoped, Operation>();
    services.AddSingleton<IOperationSingleton, Operation>();
    services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));

    // OperationService depends on each of the other Operation types.
    services.AddTransient<OperationService, OperationService>();
}
```

The `IOperationSingletonInstance` service is using a specific instance with a known ID of `Guid.Empty`. It's clear when this type is in use (its GUID is all zeroes).

The sample app demonstrates object lifetimes within and between individual requests. The sample app's `IndexModel` requests each kind of `IOperation` type and the `OperationService`. The page then displays all of the page model class's and service's `OperationId` values through property assignments:

C# Copy

```csharp
public class IndexModel : PageModel
{
    private readonly IMyDependency _myDependency;

    public IndexModel(
        IMyDependency myDependency,
        OperationService operationService,
        IOperationTransient transientOperation,
        IOperationScoped scopedOperation,
        IOperationSingleton singletonOperation,
        IOperationSingletonInstance singletonInstanceOperation)
    {
        _myDependency = myDependency;
        OperationService = operationService;
        TransientOperation = transientOperation;
        ScopedOperation = scopedOperation;
        SingletonOperation = singletonOperation;
        SingletonInstanceOperation = singletonInstanceOperation;
    }

    public OperationService OperationService { get; }
    public IOperationTransient TransientOperation { get; }
    public IOperationScoped ScopedOperation { get; }
    public IOperationSingleton SingletonOperation { get; }
    public IOperationSingletonInstance SingletonInstanceOperation { get; }
    public async Task OnGetAsync()
    {
        await _myDependency.WriteMessage(
            "IndexModel.OnGetAsync created this message.");
    }
}
```

Two following output shows the results of two requests:

First request:

Controller operations:

Transient: d233e165-f417-469b-a866-1cf1935d2518
Scoped: 5d997e2d-55f5-4a64-8388-51c4e3a1ad19
Singleton: 01271bc1-9e31-48e7-8f7c-7261b040ded9
Instance: 00000000-0000-0000-0000-000000000000

`OperationService` operations:

Transient: c6b049eb-1318-4e31-90f1-eb2dd849ff64
Scoped: 5d997e2d-55f5-4a64-8388-51c4e3a1ad19
Singleton: 01271bc1-9e31-48e7-8f7c-7261b040ded9
Instance: 00000000-0000-0000-0000-000000000000

Second request:

Controller operations:

Transient: b63bd538-0a37-4ff1-90ba-081c5138dda0
Scoped: 31e820c5-4834-4d22-83fc-a60118acb9f4

Singleton: 01271bc1-9e31-48e7-8f7c-7261b040ded9
Instance: 00000000-0000-0000-0000-000000000000

`OperationService` operations:

Transient: c4cbacb8-36a2-436d-81c8-8c1b78808aaf
Scoped: 31e820c5-4834-4d22-83fc-a60118acb9f4
Singleton: 01271bc1-9e31-48e7-8f7c-7261b040ded9
Instance: 00000000-0000-0000-0000-000000000000

Observe which of the `OperationId` values vary within a request and between requests:

- *Transient* objects are always different. The transient `OperationId` value for both the first and second client requests are different for both `OperationService` operations and across client requests. A new instance is provided to each service request and client request.
- *Scoped* objects are the same within a client request but different across client requests.
- *Singleton* objects are the same for every object and every request regardless of whether an `Operation` instance is provided in `Startup.ConfigureServices`.

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 obtain a context for the `MyScopedService` in `Program.Main`:

C# Copy

```csharp
using System;
using System.Threading.Tasks;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;

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

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

            try
            {
                var serviceContext = services.GetRequiredService<MyScopedService>();
                // Use the context here
            }
            catch (Exception ex)
            {
                var logger = services.GetRequiredService<ILogger<Program>>();
                logger.LogError(ex, "An error occurred.");
            }
        }

        await host.RunAsync();
    }

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

Scope validation

When the app is running in the Development environment, the default service provider performs checks to verify that:

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

The root service provider is created when BuildServiceProvider is called. The root service provider's lifetime corresponds to the app/server'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 app/server is shut down. Validating service scopes catches these situations when `BuildServiceProvider` is called.

For more information, see ASP.NET Core Web Host.

Request Services

The services available within an ASP.NET Core request from `HttpContext` are exposed through the HttpContext.RequestServices collection.

Request Services represent the services configured and requested as part of the app. When the objects specify dependencies, these are satisfied by the types found in `RequestServices`, not `ApplicationServices`.

Generally, the app shouldn't use these properties directly. Instead, request the types that classes require via class constructors and allow the framework inject the dependencies. This yields classes that are easier to test.

Note

Prefer requesting dependencies as constructor parameters to accessing the `RequestServices` collection.

Design services for dependency injection

Best practices are to:

- Design services to use dependency injection to obtain their dependencies.
- Avoid stateful, static classes and members. Design apps to use singleton services instead, which avoid creating global state.
- Avoid direct instantiation of dependent classes within services. Direct instantiation couples the code to a particular implementation.
- Make app classes small, well-factored, and easily tested.

If a class seems to have too many injected dependencies, this is generally a sign that the class has too many responsibilities and is violating the Single Responsibility Principle (SRP). Attempt to refactor the class by moving some of its responsibilities into a new class. Keep in mind that Razor Pages page model classes and MVC controller classes should focus on UI concerns. Business rules and data access implementation details should be kept in classes appropriate to these separate concerns.

Disposal of services

The container calls Dispose for the IDisposable types it creates. If an instance is added to the container by user code, it isn't disposed automatically.

C# Copy

```csharp
// Services that implement IDisposable:
public class Service1 : IDisposable {}
public class Service2 : IDisposable {}
public class Service3 : IDisposable {}

public interface ISomeService {}
public class SomeServiceImplementation : ISomeService, IDisposable {}
public void ConfigureServices(IServiceCollection services)
{
    // The container creates the following instances and disposes them automatically:
    services.AddScoped<Service1>();
    services.AddSingleton<Service2>();
    services.AddSingleton<ISomeService>(sp => new SomeServiceImplementation());

    // The container doesn't create the following instances, so it doesn't dispose of
    // the instances automatically:
    services.AddSingleton<Service3>(new Service3());
    services.AddSingleton(new Service3());
}
```

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 the built-in container doesn't support, such as:

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

The following 3rd party containers can be used with ASP.NET Core apps:

- Autofac
- DryIoc
- 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 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 once by a single thread.

Recommendations

- `async/await` and `Task` based service resolution is not supported. C# does not support asynchronous constructors; therefore, the recommended pattern is to 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 some other object. It's better to request the actual item via DI.

- Avoid static access to services (for example, statically-typing IApplicationBuilder.ApplicationServices for use elsewhere).

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

  Incorrect:

  C# Copy

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

        ...
    }
}
```

Correct:

C#

Copy

```csharp
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).

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.

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
- App startup in ASP.NET Core
- Factory-based middleware activation 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