Dependency Injection

.NET CORE

A *dependency* is any object that, to function, another object requires. The ***Dependency injection (DI)*** design pattern is a technique for achieving ***Inversion of Control (IoC)*** between classes and their dependencies.

HTTPS://DOCS.MICROSOFT.COM/EN-US/ASPNET/CORE/FUNDAMENTALS/DEPENDENCY-INJECTION?VIEW=ASPNETCORE-3.1

Dependencies – Overview

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1

An instance of the *MyDependency* class can be created in another class to make the *WriteMessage* method available to that class.

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

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

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

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

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

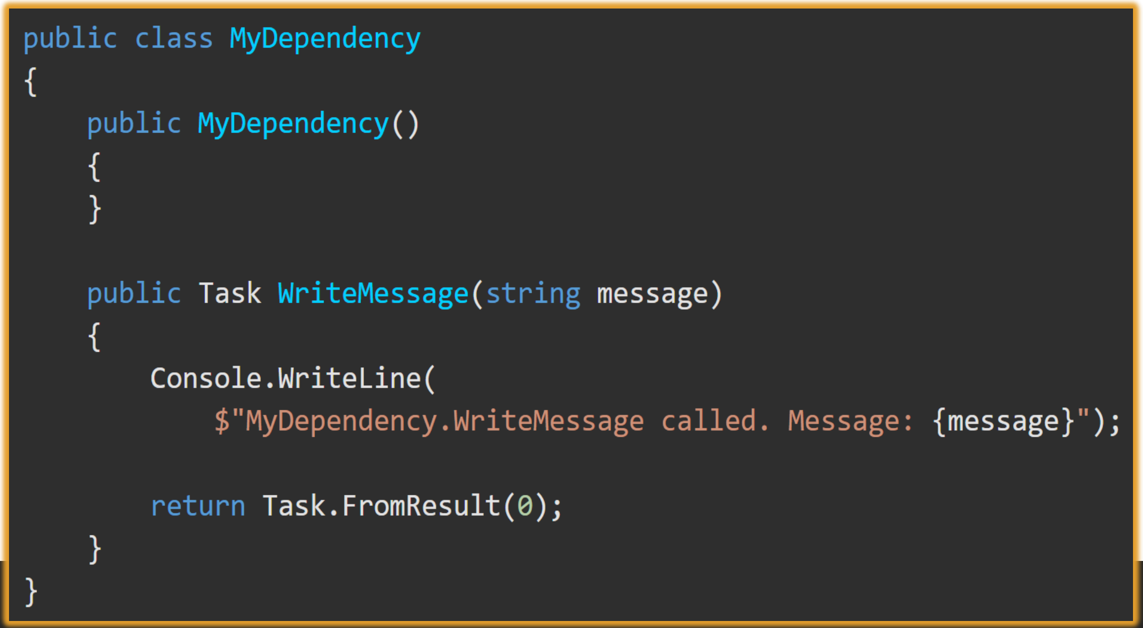
MyDependency class is a dependency of *IndexModel* class.

Dependency Inversion – Overview

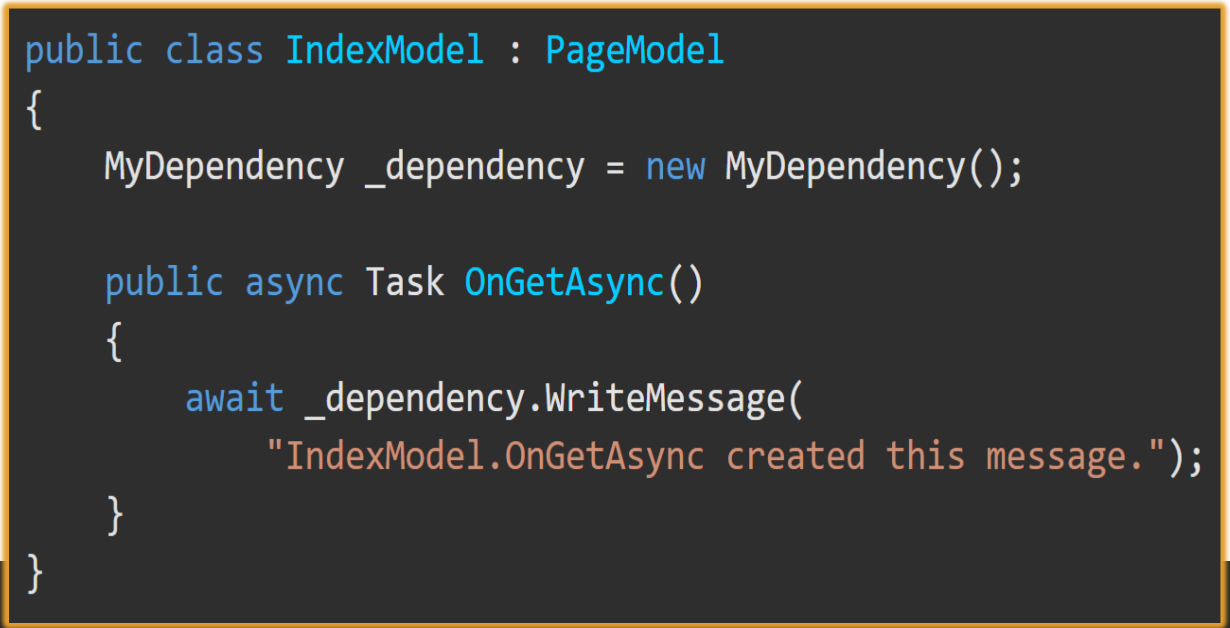
https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1

Code dependencies like this are problematic and should be avoided.

- To replace *MyDependency* with a different implementation, the class must be modified.
- If *MyDependency* has dependencies, they must be configured by the class.
- This implementation is difficult to unit test.



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

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

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



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

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

Dependency Injection – A better way

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1

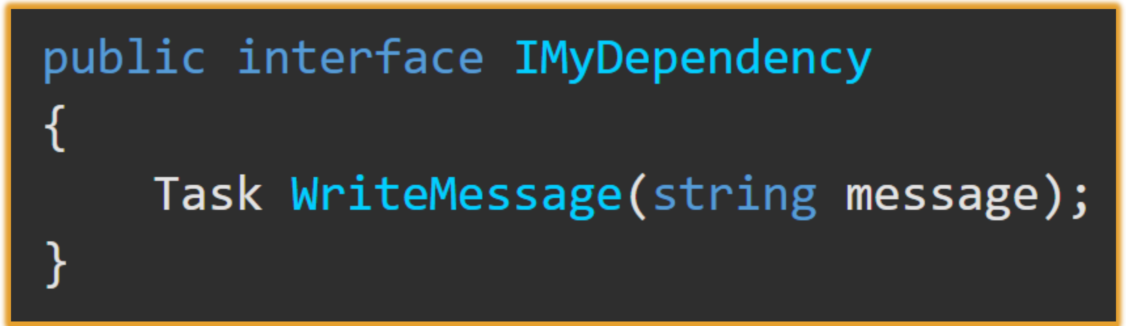
Dependency injection addresses service dependency problems by:

- providing (through ASP.NET Core) a built-in ***service container*** called *IServiceProvider*.
- registering dependencies in the ***service container.***
- registering services through the app's ***Startup.ConfigureServices*** (in Startup.cs) method.
- using an interface (or base class) to abstract the dependency implementation.
- injecting 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.

Dependency Injection – Step by Step(1)

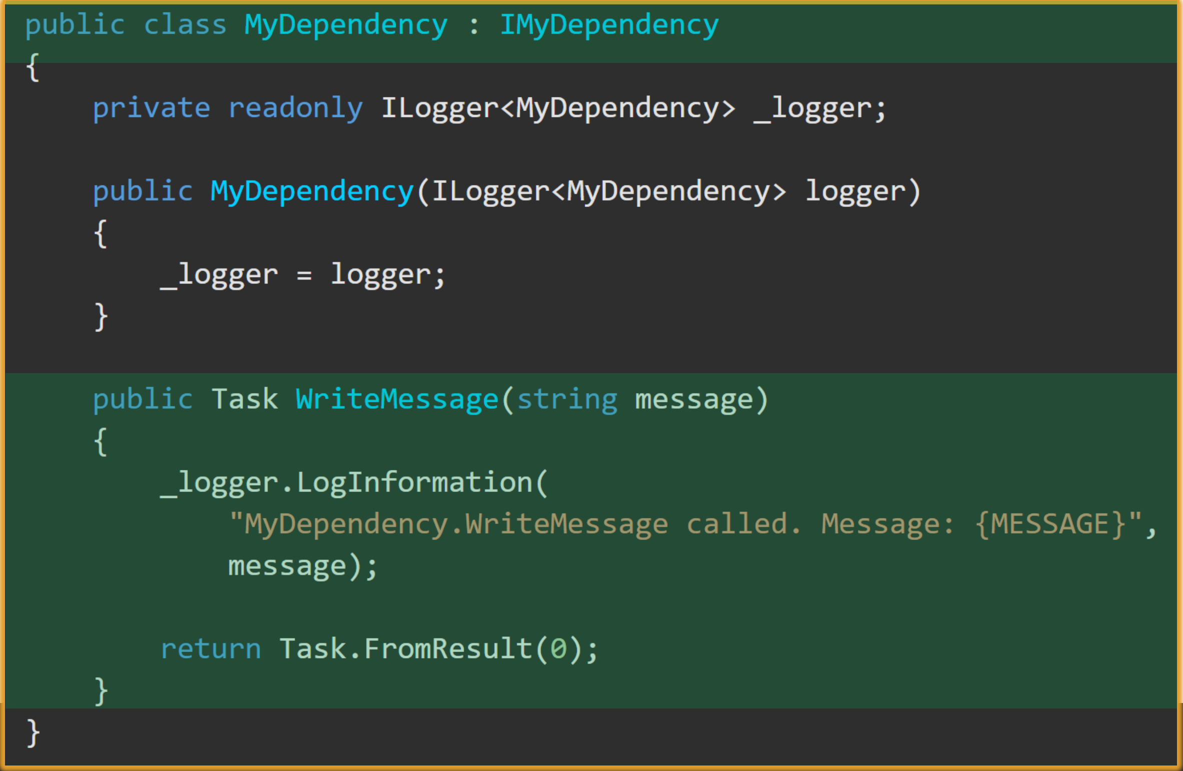
https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1



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

1) Create an interface where you declare a method that you want to make available through Dependency Injection.

2)Define the method in a class that implements the Interface.



```
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);
    }
}
```

Dependency Injection – Step by Step(2)

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1

```
public void ConfigureServices(IServiceCollection services)
{


    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>();
}
```

3)Add the dependency to ***ConfigureServices*** with ***services.[desiredScope]<[interface], [class]>();***

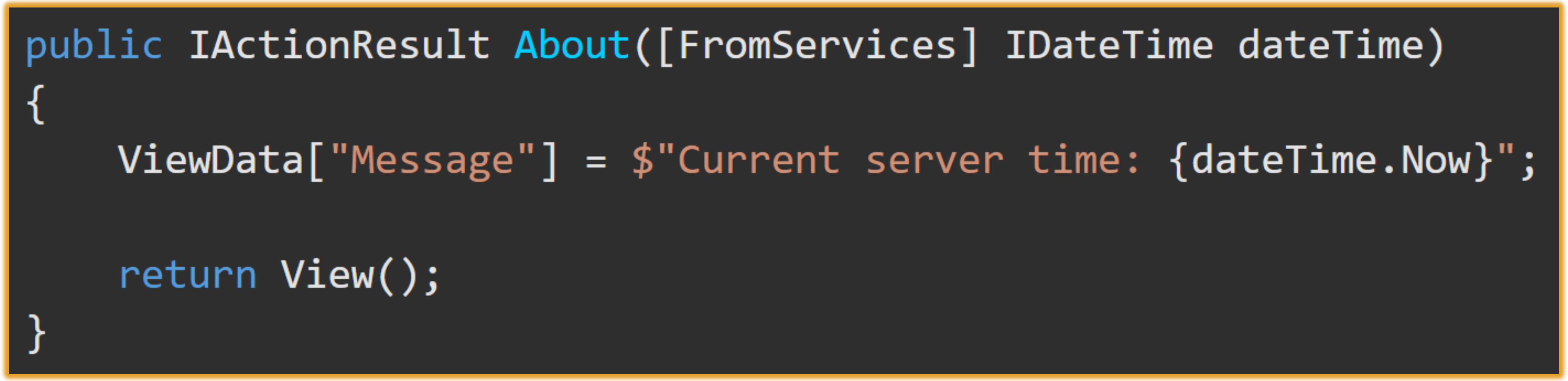
4) Inject the dependency into the constructor of the dependent class and assign it to a ***private*** variable of the ***interface*** type.

```
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.");
    }
}
```

Dependency Injection – [FromServices] Alternative to Constructor Injection

https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/dependency-injection?view=aspnetcore-3.1#action-injection-with-fromservices

After registering a service with the ***service container***, the *[FromServices]* attribute enables injecting the registered service directly into an ***action method*** without using constructor injection in the ***Controller***.

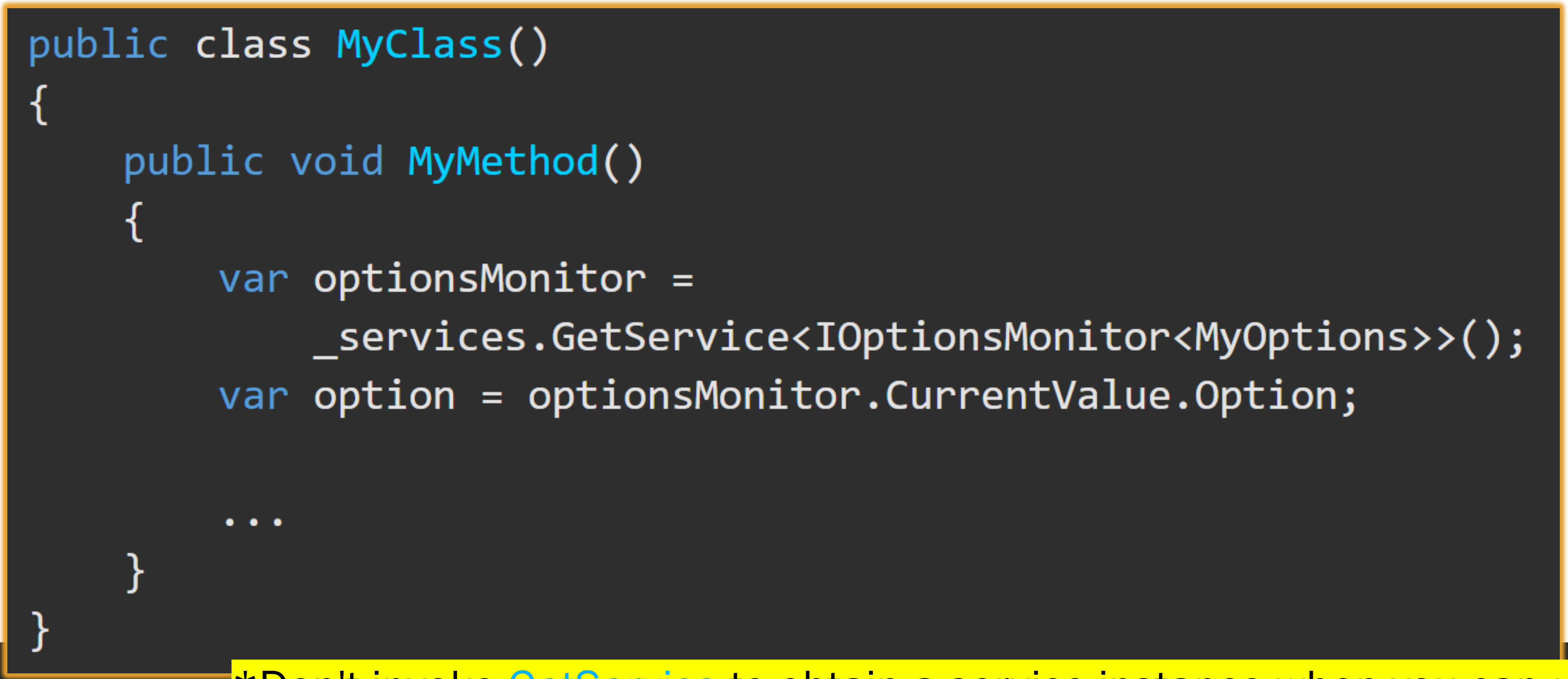


```
public IActionResult About([FromServices] IDateTime dateTime)
{
    ViewData["Message"] = $"Current server time: {dateTime.Now}";

    return View();
}
```

Dependency Injection – .GetService<>() Alternative to Constructor Injection

https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/dependency-injection?view=aspnetcore-3.1#action-injection-with-fromservices

When you are unable to obtain an instance of a needed service by **Dependency Injection, *.GetService*<>** can be used to get a service object.



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

        ...
    }
}
```

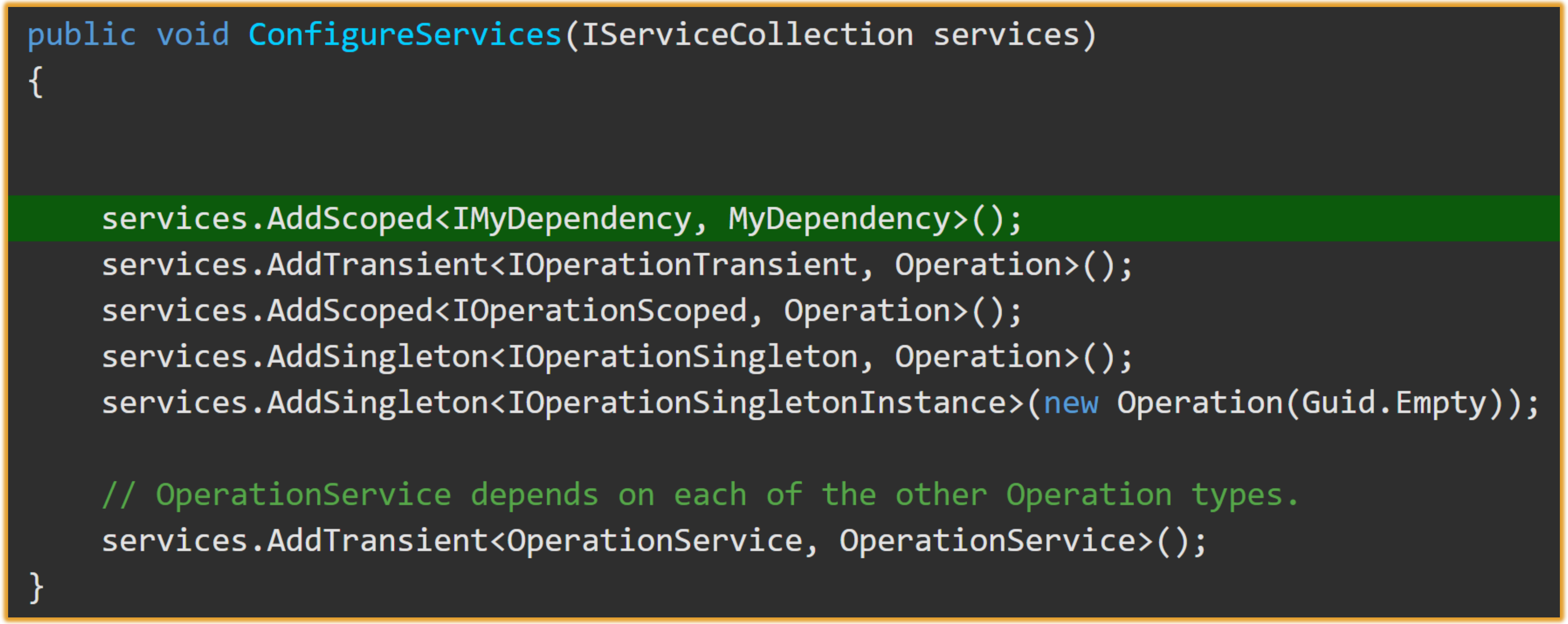
*Don't invoke GetService to obtain a service instance when you can use DI instead.

Service Type Lifetimes

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1#transient

Service	Description
Transient	*Transient* services (AddTransient) are created each time they're requested from the service container. Best for lightweight, stateless services.
Scoped	*Scoped* lifetime services (AddScoped) are created once per HTTP request (connection).
Singleton	*Singleton* services (AddSingleton) are created the first time they're requested (or when *Startup.ConfigureServices* is run). Every subsequent request uses the same instance.

Dependency Injection - Scopes Examples.

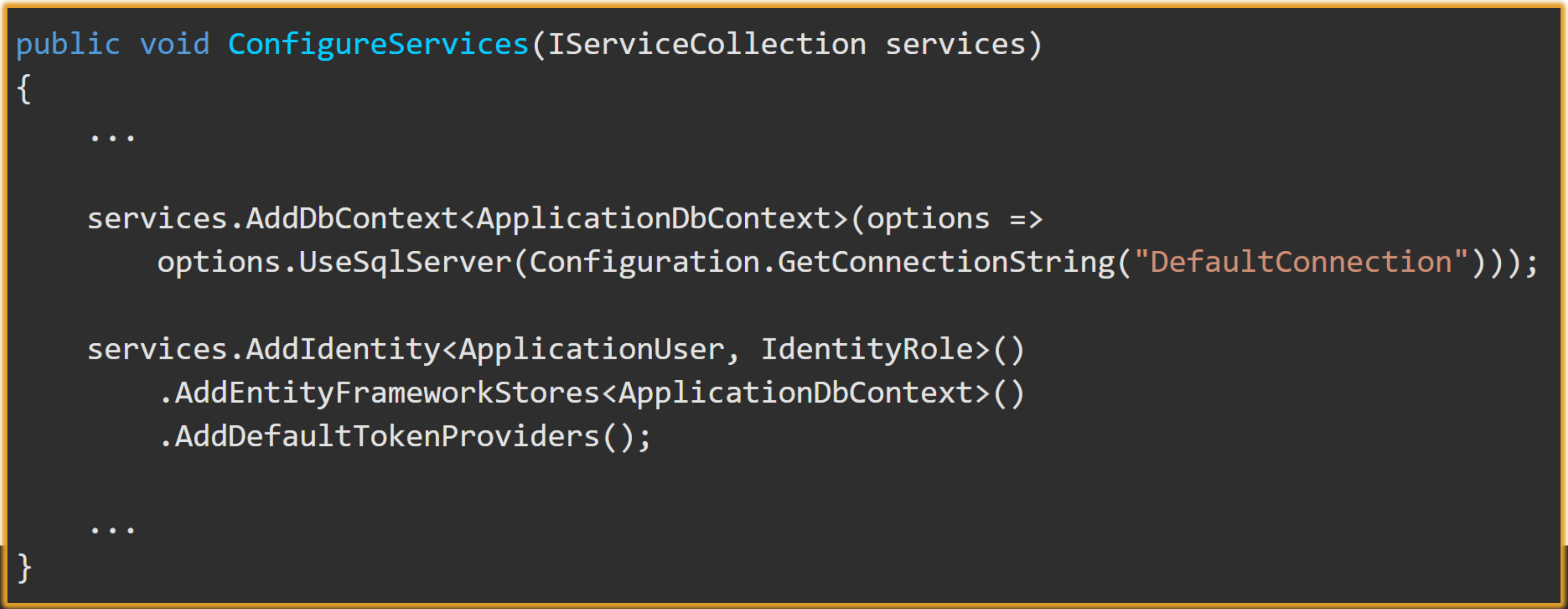


```
public void ConfigureServices(IServiceCollection services)
{


    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>();
}
```

Dependency Injection - .addDbContext

https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.entityframeworkservicecollectionextensions.adddbcontext?view=efcore-3.1
https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1#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.



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

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

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

    ...
}
```

Dependency Injection – Best Practices

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1#design-services-for-dependency-injection

Best design 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 classes small, well-factored, and easily tested.
- If a class seems to have too many injected dependencies, it's a sign that the class has too many responsibilities and is violating the *Single Responsibility Principle (SRP)*.

(Cont.) Dependency Injection – Best Practices

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1#design-services-for-dependency-injection
https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1#recommendations

- When services are resolved by ***IServiceProvider***, constructor injection requires a public constructor.
- The built-in ***service container*** is designed to serve the needs of the framework and most consumer apps. It is recommended to use the built-in ***container*** unless you need a specific feature that the built-in ***container*** doesn't support.
- ***Dependency Injection*** is an <u>alternative</u> 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.