3.Dependency Injection

❖ Dependency injection

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

Examine the following MessageWriter class with a Write method that other classes depend on:

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

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

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

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

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

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

Dependency injection addresses these problems through:

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

```
using DependencyInjection.Example;

var builder = Host.CreateDefaultBuilder(args);

builder.ConfigureServices(
    services =>
        services.AddHostedService<Worker>()
            .AddScoped<IMessageWriter, MessageWriter>());

using var host = builder.Build();

host.Run();
```

❖ Built-in IoC Container

- A DI Container/ IoC container is a framework to create dependencies and inject them automatically when required.
- It automatically creates objects based on the request and injects them when required.
- DI Container helps us to manage dependencies within the application in a simple and easy way.

- The DI container creates an object of the defined class and also injects all the required dependencies as an object a constructor, a property, or a method that is triggered at runtime and disposes itself at the appropriate time.
- This process is completed so that we don't have to create and manage objects manually all the time.
- ASP.NET Core framework includes built-in IoC container for automatic dependency injection.
- The built-in IoC container is a simple yet effective container.

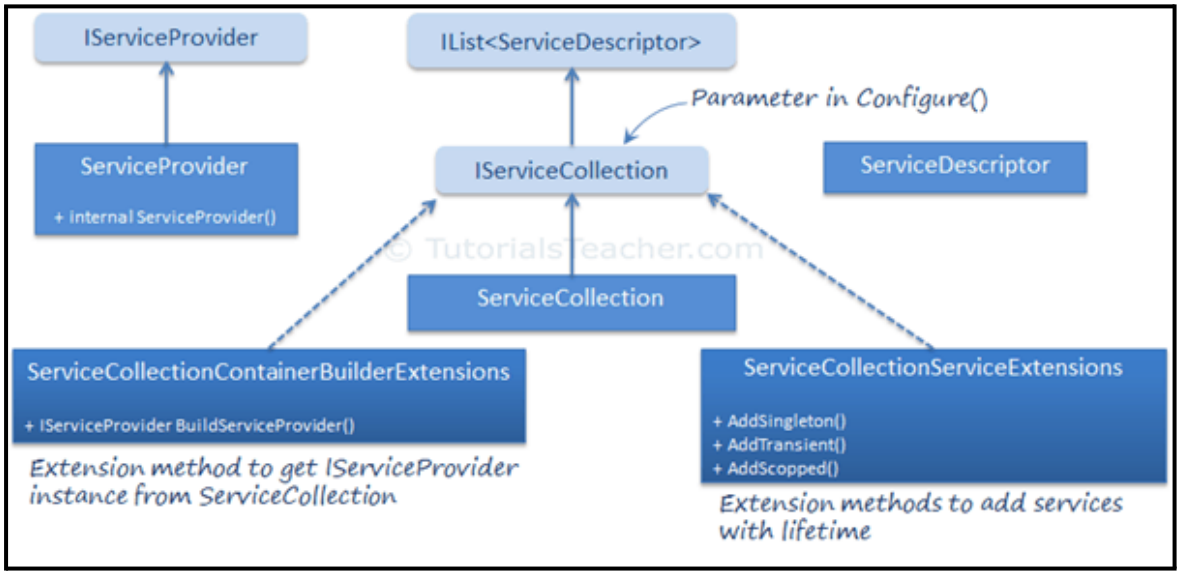
The followings are important interfaces and classes for built-in IoC container:

Interfaces:

1. IServiceProvider
2. IServiceCollection

Classes:

1. ServiceProvider
2. ServiceCollection
3. ServiceDescription
4. ServiceCollectionServiceExtensions
5. ServiceCollectionContainerBuilderExtensions



- **IServiceCollection**

We can register application services with built-in IoC containers in the Configure method of Startup class by using IServiceCollection. IServiceCollection interface is an empty interface. It just inherits IList<servicedescriptor>. See the source code here.

The ServiceCollection class implements IServiceCollection interface. See the ServiceCollection source code here.

So, the services you add in the IServiceCollection type instance, it actually creates an instance of ServiceDescriptor and adds it to the list.

```
public void ConfigureServices(IServiceCollection services)
{
    var serviceProvider = services.BuildServiceProvider();

}
```

- **IServiceProvider**

IServiceProvider is an interface in the .NET Core and .NET 5+ frameworks that defines a mechanism for retrieving services from a service container. It is part of the built-in dependency injection system in ASP.NET Core and other .NET applications.

IServiceProvider includes the GetService method. The ServiceProvider class implements IServiceProvider interface which returns registered services with the container. We cannot instantiate ServiceProvider class because its constructors are marked with internal access modifiers.

The IServiceProvider interface typically has methods like GetService and GetRequiredService for retrieving services by their type. These methods allow components of an application to request instances of services that have been registered with the service container during application startup.

Here's a brief overview of these methods:

GetService: This method retrieves a service of the specified type from the service container. If the service is not found, **it returns null**.

GetRequiredService: This method retrieves a service of the specified type from the service container. If the service is not found, **it throws an exception**.

- **ServiceCollectionServiceExtensions**

The ServiceCollectionServiceExtensions class includes extension methods related to service registrations which can be used to add services with lifetime. AddSingleton, AddTransient, AddScoped extension methods defined in this class.

- **ServiceCollectionContainerBuilderExtensions**

ServiceCollectionContainerBuilderExtensions class includes BuildServiceProvider extension method which creates and returns an instance of ServiceProvider.

❖ Registering Application Service

- register the interface mapping with the DI (Dependency Injection) container.
- Go to the Startup.cs file and then add the following code within the ConfigureServices() method to Registering application service.

```
public void ConfigureServices(IServiceCollection services)
{
services.AddTransient<IMusicManager, MusicManager>();
//register other services here
}
```

- The preceding code registers the IMusicManager interface as the service type and maps the MusicManager concrete class as the implementation type in the DI container.
- This tells the framework to resolve the required dependency that has been injected into the HomeController class constructor at runtime.
- The beauty of DI is that it allows you to change whatever component that you want for as long as it implements the interface.
- What this means is that you can always replace the MusicManager class mapping to something else for as long as it implements the IMusicManager interface without impacting the HomeController implementation.
- The ConfigureServices() method is responsible for defining the services that the application uses, including platform features, such as Entity Framework Core, authentication, your own service, or even third-party services.
- Initially, the IServiceCollection interface provided to the ConfigureServices() method has services defined by the framework, including Hosting, Configuration, and Logging.

- **Benefits of DI**
  - It promotes the loose coupling of components.
  - It helps in separation of concerns.
  - It promotes the logical abstractions of components.
  - It facilitates unit testing.
  - It promotes clean and more readable code, which makes code maintenance manageable.

❖ Understanding Service Lifetime

When we register services in a container, we need to set the lifetime that we want to use. The service lifetime controls how long a result object will live for after it has been created by the container. The lifetime can be created by using the appropriate extension method on the IServiceCollection when registering the service.

There are three lifetimes that can be used with Microsoft Dependency Injection Container, they are:

Transient — Services are created each time they are requested. It gets a new instance of the injected object, on each request of this object. For each time you inject this object is injected in the class, it will create a new instance.

Scoped — Services are created on each request (once per request). This is most recommended for WEB applications. So for example, if during a request you use the same dependency injection, in many places, you will use the same instance of that object, it will make reference to the same memory allocation.

Singleton — Services are created once for the lifetime of the application. It uses the same instance for the whole application.

The dependency injection container keeps track of all instances of the created services, and they are disposed of or released for garbage collector once their lifetime has ended. This is how we can configure the DI in .NET core:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IMyDependencyA, MyDependencyA>();
    services.AddSingleton<IMyDependencyB, MyDependencyB>();
    services.AddScoped<IMyDependencyC, MyDependencyC>();
}
```

To have a better understanding of the difference between these three lifetimes, we will see a code that uses these three lifetimes and we will see the difference between them.

```
using System;
using DependencyInjectionAndServiceLifetimes.Interfaces;

namespace DependencyInjectionAndServiceLifetimes.Services
{
    public class ExampleScopedService : IExampleScopedService
    {
        private readonly Guid Id;

        public ExampleScopedService()
        {
            Id = Guid.NewGuid();
        }

        public string GetGuid() => Id.ToString();
    }
}
```

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(99));
	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(99));
	No	Yes	Yes

Method	Automatic object disposal	Multiple implementations	Pass args

AddSingleton(new {IMPLEMENTATION})
 	
Examples:
 	
services.AddSingleton(new MyDep());
services.AddSingleton(new MyDep(99));
	No	No	Yes

❖ Extension Methods for Registration

ASP.NET Core framework includes extension methods for each types of lifetime; AddSingleton(), AddTransient() and AddScoped() methods for singleton, transient and scoped lifetime respectively.

The following example shows the ways of registering types (service) using extension methods.

```
namespace AspDotNetCoreApi
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.ConfigureCors(Configuration)
                .ConfigureSwagger()
                .ConfigureVersioning()
                .ConfigureSetting(Configuration)
                .ConfigureAuthentication(Configuration);
            // Service injections
            services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
            services.AddSingleton<IDapperHelper, DapperHelper>();
            services.AddHttpClient();

            services.AddScoped<IAdventureWorksRepository, AdventureWorksRepository>();
            services.AddScoped<IAdventureWorksOrchestrator, AdventureWorksOrchestrator>();
            services.AddControllers();
            //other code
        }
    }
}
```

For more Details see Reference section.

❖ Constructor injection

- This approach basically allows you to inject lower-level dependent components into your class by passing them into the constructor class as arguments.
- This approach is the most commonly used when building ASP.NET Core applications. when you create an ASP.NET Core project from the default template, you will see that DI is, by default, integrated.
- You can verify this yourself by looking into the HomeController class and you should see the ILogger interface being injected into the class constructor, as shown in the following code:

```
public class HomeController : Controller
{
private readonly ILogger<HomeController> _logger;
public HomeController(ILogger<HomeController> logger)
{
_logger = logger;
}
}
```

The ILogger interface is registered by the logging abstraction's infrastructure and is registered by default in the framework as a Singleton.

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

The preceding code registers the service as a Singleton and uses the generic open types technique. This allows the DI container to resolve dependencies without having to explicitly register services with generic constructed types.

[FromService] <Repository>

```
[HttpGet("name")]
0 references
public IActionResult GetName([FromServices] IProductRepository _productRepository)
{
    var name = _productRepository.GetName();
    return Ok(name);
}
```