Dependency Injection

This module talks about Dependency Injection (DI) in the context of ASP.NET Core. Moreover, this module will get you up to speed with the concept of DI, its capabilities, and how it is used in ASP.NET Core applications. We will review the different types of DI by following code examples so that you will be able to understand how and when to apply them in situations where they may be required. We will also be looking at DI containers, service lifetimes, and how to handle complex scenarios as you progress throughout the module. By the end of this module, you'll be able to understand how DI works by following some practical examples. You should then be able to apply the knowledge and skills that you have learned to build real-world and powerful ASP.NET Core applications, and take advantage of the benefits that DI has to offer.

Here is the list of topics that we will be covering in this module:

- Learning dependency injection in ASP.NET Core
- Reviewing types of dependency injection
- Understanding dependency injection containers
- Understanding dependency lifetimes
- Handling complex scenarios

Technical requirements

This module contains code snippets written in C# for demonstrating various scenarios. Please verify that you have installed the required software and tools listed in Module 1, Introduction to ASP.NET Core 5.

Before diving into this module, make sure that you read the first two modules so that you have a basic understanding of ASP.NET Core and C# in general, and how each of them works together.

If you're ready, let's jump right into it.

Learning dependency injection in ASP.NET Core

To give you a bit of a background, before .NET Core came into being, the only way to get DI in your applications was through the use of third-party frameworks such as Autofac, LightInject, Unity, and many others. The good news is that DI is now treated as a firstclass citizen in ASP.NET Core. This simply means that you don't need to do much to make it work.

The built-in Microsoft DI container does have its limitations though. For example, the default DI doesn't provide advanced capabilities, such as property injection decorators, injections based on name, child containers, convention-based registration, and custom lifetime management. So, if you find features that are not available in the default DI container, then that's when you'll need to consider looking at some other third-party DI frameworks mentioned earlier as an alternative. However, it is still recommended to use the default DI framework for building ASP.NET Core applications that don't require you to implement any specific features. This will lessen your application package dependencies

and make your code cleaner and more manageable without having to rely on third-party frameworks. The .NET Core team did a pretty good job of providing us with the most common features and you probably won't need anything else.

In this section, we'll do some hands-on coding for you to enable you to better understand the advantages and benefits of DI. We'll start by looking at a common problem and then apply DI to resolve the problem.

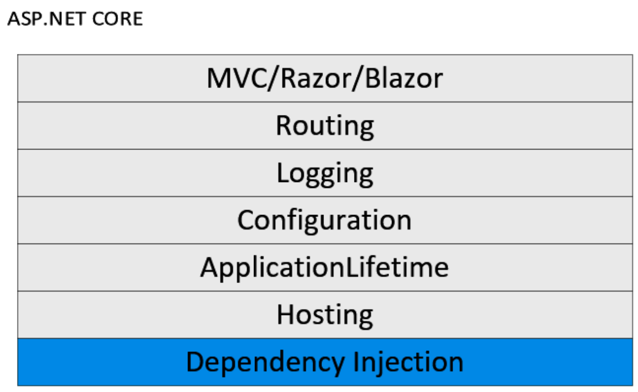
Understanding what DI is

There is a plethora of information on the web that defines DI, but a simple definition is as

follows:

> *"Dependency injection is a design pattern that enables developers to write loosely coupled code."*

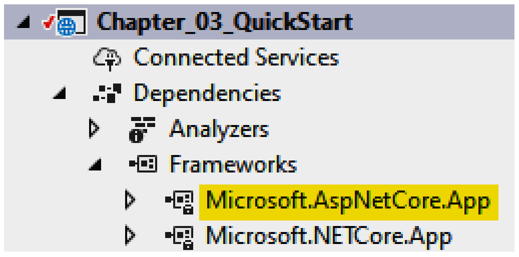
In other words, DI helps you to write clean and more maintainable code by solving dependency problems. DI makes it easy to mock object dependencies for unit testing and makes your application more flexible by swapping or replacing dependencies without having to change the consuming classes. In fact, the core foundation of ASP.NET Core frameworks relies heavily on DI, as shown in the following diagram:



All framework-provided services, such as **Hosting, Configuration, ApplicationLifetime, Logging, Routing**, and many others use DI under the hood, and they are, by default, registered to the DI container when the application web host is built.

The default DI in .NET Core sits under the Microsoft.Extensions.DependencyInjection namespace, whose implementation is packed into a separate NuGet package (you can learn more at https://www.nuget.org/packages/Microsoft.Extensions.DependencyInjection/).

When you create an ASP.NET Core application from the default template, the application references the Microsoft.AspNetCore.App NuGet package, as shown in the following screenshot:



Chapter_03_QuickStart
Connected Services
Dependencies
Analyzers
Frameworks
Microsoft.AspNetCore.App
Microsoft.NETCore.App

This assembly provides a set of APIs, including the Microsoft.Extensions.DependencyInjection assembly for building ASP.NET Core applications.

The ASP.NET team designed the DI framework separately so that you will still be able to leverage its features outside ASP.NET Core applications. What this means is that you will be able to use DI in event-driven cloud apps such as Azure Functions and AWS Lambda, or even in console applications.

The use of DI mainly supports the implementation of the following two related concepts:

- **Dependency Inversion Principle (DIP):** This is a software design principle and represents the "D" in the SOLID principles of object-oriented programming.
  It provides a guideline for avoiding a dependency risk and solving common dependency problems. However, this principle doesn't state any specific technique for you to implement.
- **Inversion of Control (IoC):** This is a technique that follows the DIP guidelines. This concept is the process of creating application components in a detached state, preventing higher-level components from having direct access to lower-level components, and allowing them to only interact via abstractions.

DI is an implementation technique that follows the concept of IoC. It enables you to access lower-level components from a higher-level component through component injections.

DI follows two SOLID principles: DIP and the **Single Responsibility Principle (SRP)**.

These concepts are crucial for creating well-designed and well-decoupled applications, and you should consider applying them in any situation where required. Check out the Further reading section at the end of this module to learn more about the SOLID principles.

You may have heard these terms and concepts and you still find them very confusing.

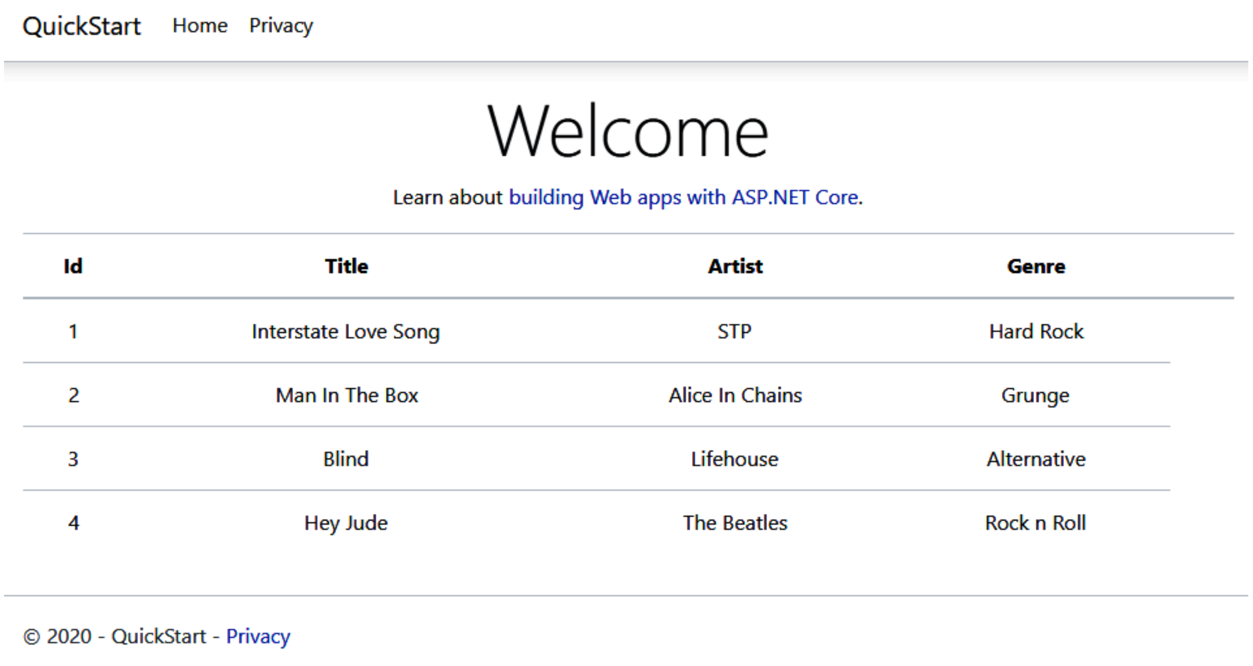
Well, here is an analogy that might help you better understand them.

Let's say you are making your own song and you wanted to upload it on the web so that your friends can watch and hear it. You can think of the DIP as a way to record music.

It doesn't matter how you record the song. You could use a video recorder, a camera, a smartphone, or a studio recorder. IoC is choosing how you would actually record your music and polish it with the help of some tools. For example, you can use a combination of audio and camera recorders to record your song. Typically, they are recorded as raw files. You would then use an editor tool to filter and polish the raw files to come up with a great output. Now, if you wanted to add some effects, text visualization, or graphics background, then that's where DI comes into play. It allows you to inject whatever files your file depends on to generate the output you expect. Keep in mind that in this analogy, both IoC and DI rely on using the editor tool to generate the ultimate output (high-level component) based on raw files (low-level component). In other words, both IoC and DI refer to the same concept by using the editor tools to improve your video output. To illustrate this, let's look at a brief example.

The common dependency problem

Consider we have the following page that displays a list of music in a typical MVC web application:



Welcome

Learn about building Web apps with ASP.NET Core.

Id	Title	Artist	Genre
1	Interstate Love Song	STP	Hard Rock
2	Man In The Box	Alice In Chains	Grunge
3	Blind	Lifehouse	Alternative
4	Hey Jude	The Beatles	Rock n Roll

© 2020 - QuickStart - Privacy

Let's break down how we came up with the result shown in the previous screenshot. For your quick reference, here's the class called MusicManager, which exposes a method for obtaining the list of music:

```
using DI.Models;
using System.Collections.Generic;
namespace DI.DataManager{
  public class MusicManager{
    public List<SongModel> GetAllMusic() {
      return new List<SongModel>{
        new SongModel { Id = 1, Title = "Interstate Love Song",
                        Artist ="STP",Genre = "Hard Rock"},
        new SongModel { Id = 2, Title = "Man In The Box",
                        Artist ="Alice In Chains", Genre = "Grunge"},
        new SongModel { Id = 3, Title = "Blind",
                        Artist="Lifehouse", Genre = "Alternative"},
        new SongModel { Id = 4, Title = "Hey Jude",
                        Artist ="The Beatles", Genre = "Rock n Roll"}
      };
    }
  }
}
```

The preceding code is nothing but a plain class that contains a method, GetAllMusic(). This method is responsible for returning all music entries from the list. The implementation could vary depending on your data store, and you could be pulling them from a database or via an API call. However, for this example, we just return a static list of data for simplicity's sake.

The SongModel class lives inside the Models folder with the following structure:

```
namespace DI.Models {
  public class SongModel {
    public int Id { get; set; }
    public string Title { get; set; }
    public string Artist { get; set; }
    public string Genre { get; set; }
  }
}
```

Nothing fancy. The preceding code is just a dumb class that houses some properties that the View expects.

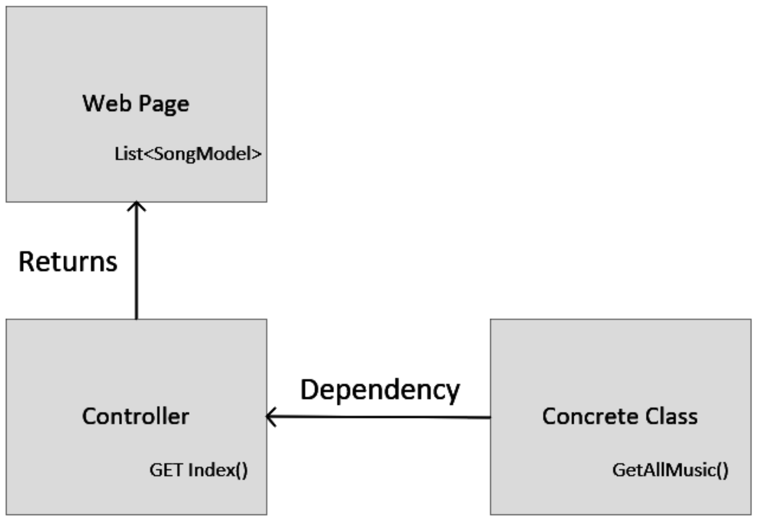
Without DI, we would normally call a method from a class directly into the Controller class to render View, as shown in the following code block:

```
public IActionResult Index() {
  MusicManager musicManager = new MusicManager();
  var songs = musicManager.GetAllMusic();
  return View(songs);
}
```

The Index() method in the preceding code will be invoked when you perform an HTTP GET request. The method is responsible for rendering the data into the View. You can see that it creates an instance of the MusicManager class by invoking the new operator.

This is known as a "dependency" because the Index() method is now dependent on the MusicManager object for fetching the required data.

Here is a high-level graphical representation of what the code logic is doing:



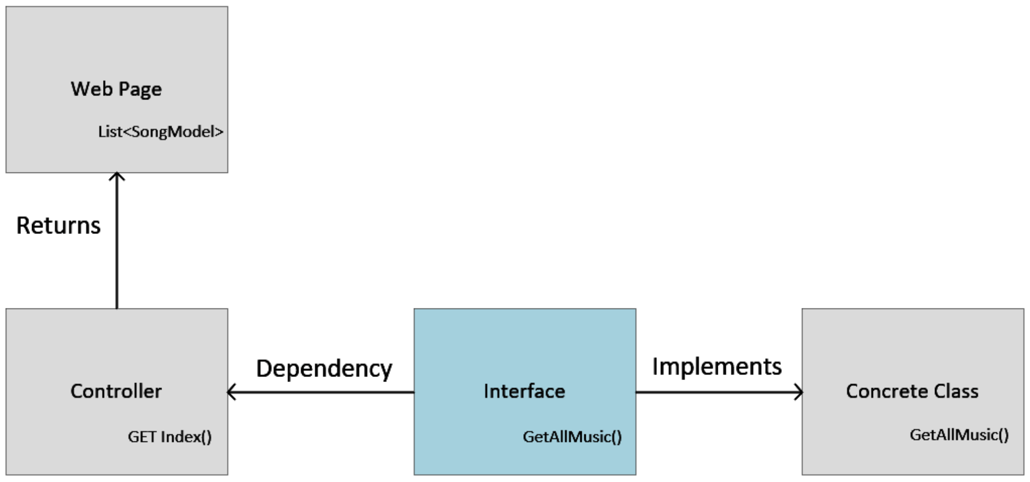
In the preceding diagram, the Controller box represents the higher-level component where it refers to the concrete class implementation as a direct dependency, which represents the lower-level component.

While the existing implementation works, this approach could result in making your code difficult to manage because the object is tightly coupled to the method itself. Imagine you have a bunch of methods that rely on the MusicManager object and when you rename it or change its implementation in the future, you would be forced to update all your methods that depend on that object, which could be harder to maintain and problematic when it comes to unit testing your Controllers. Be aware that refactoring bad code can be time-consuming and expensive, so it is better do it correctly from the outset.

The ideal approach for avoiding such a mess is to clean up our code and take advantage of using interfaces and DI.

Making use of DI

To resolve the dependency problem that our HomeController had, we need to do a little bit of code refactoring. Here's a graphical illustration of the goal that we are aiming for:



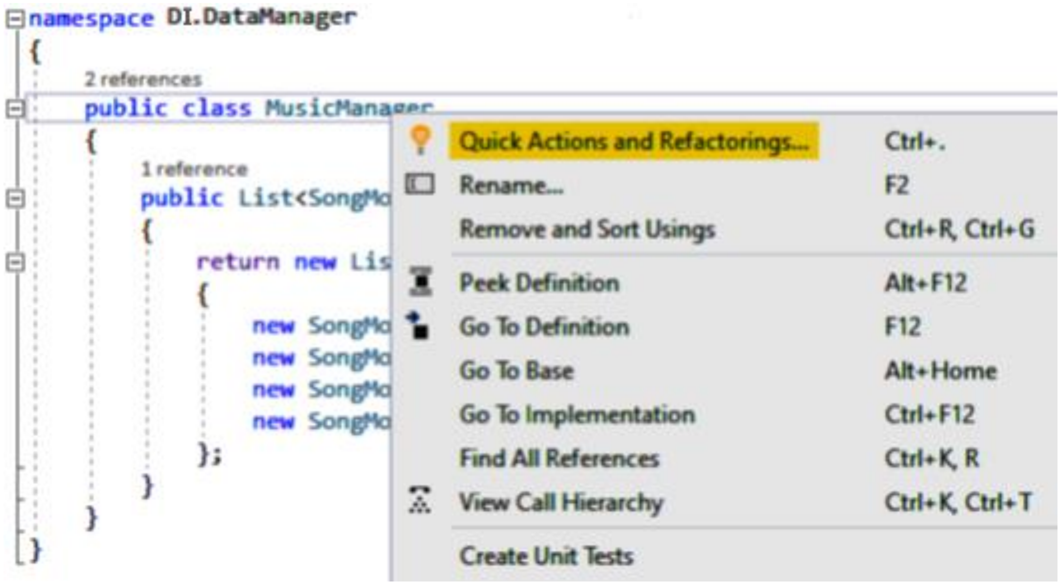
As you can see from the preceding diagram, we just need to create an interface to resolve the dependency problem. This approach avoids the direct dependency to the lower-level component and instead, it creates an abstraction that both components depend on.

This now makes the Controller class more testable and extensible, and makes the application more maintainable.

Let's proceed and start creating an interface. There are two ways to create an interface:

Either you create it yourself or use the built-in refactoring features provided by Visual Studio 2019. Since we already have an existing class that we wanted to extract as an interface, using the refactoring feature makes a lot of sense. To do this, you need to perform the following steps:

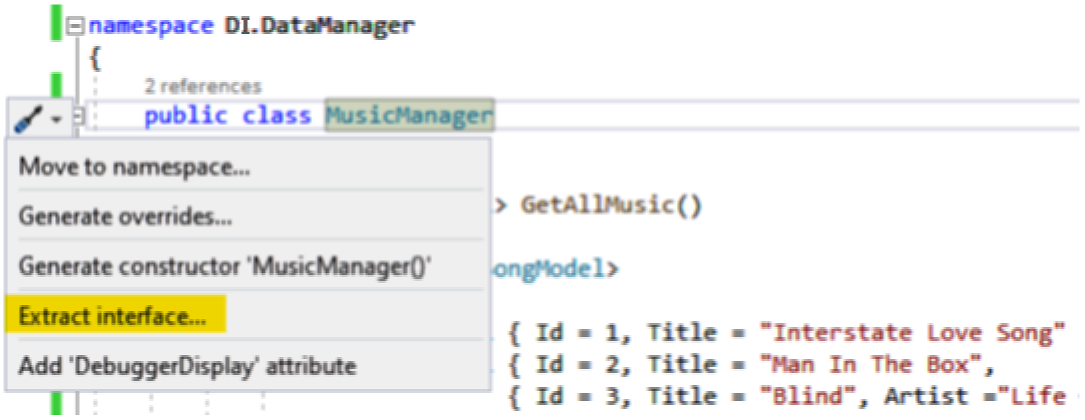
1. Just simply right-click on the MusicManager class and select **Quick Actions and Refactorings...**, as shown:



```
namespace DI.DataManager
{
    2 references
    public class MusicManager
    {
        1 reference
        public List<SongMo
        {
            return new Lis
            {
                new SongMo
                new SongMo
                new SongMo
                new SongMo
            };
        }
    }
}
```

Quick Actions and Refactorings... Ctrl+.
Rename... F2
Remove and Sort Usings Ctrl+R, Ctrl+G
Peek Definition Alt+F12
Go To Definition F12
Go To Base Alt+Home
Go To Implementation Ctrl+F12
Find All References Ctrl+K, R
View Call Hierarchy Ctrl+K, Ctrl+T
Create Unit Tests

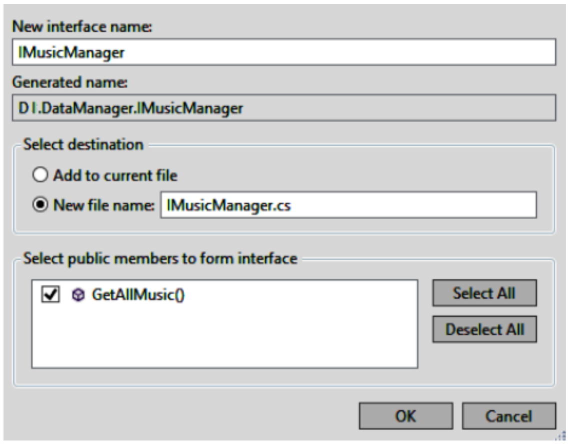
2. Then, select **Extract interface...**:



```
namespace DI.DataManager
{
    2 references
    public class MusicManager
> GetAllMusic()
ongModel>
{ Id = 1, Title = "Interstate Love Song"
{ Id = 2, Title = "Man In The Box",
{ Id = 3, Title = "Blind", Artist ="Life
```

Move to namespace...
Generate overrides...
Generate constructor 'MusicManager()'
Extract interface...
Add 'DebuggerDisplay' attribute

3. Now, you should be presented with a pop-up dialog to configure the interface, as shown in the following screenshot:



New interface name: IMusicManager

Generated name: DI.DataManager.IMusicManager

Select destination

○ Add to current file

● New file name: IMusicManager.cs

Select public members to form interface

☑ GetAllMusic()

4. You could change the default configuration if you like, but for this exercise, let's just stick with the defaults and click on OK. Here's the generated code that is created automatically by Visual Studio:

```
using DI.Models;
using System.Collections.Generic;
namespace DI.DataManager {
  public interface IMusicManager {
    List<SongModel> GetAllMusic();
  }
}
```

The preceding code is just a simple interface with the GetAllMusic() method signature that returns a type of List<SongModel>. We won't deep dive into the details of interfaces in this book, but to give you a brief overview, a couple of benefits associated with the interface are the fact that it provides abstraction to help reduce coupling in our code and enables us to provide different implementations for the method without affecting other classes.

Now, when you go back to the MusicManager class, you will see that the class has been updated to inherit the interface:

```
public class MusicManager : IMusicManager
```

Neat! With just a few clicks, Visual Studio automatically sets up everything for us.

What's left for us to do here is to refactor the HomeController class to make use of the interface and DI, and then register the interface mapping with the DI container. Let's proceed and switch back to the HomeController class and update the code so that it will look similar to this:

```
namespace DI.Controllers {
  public class HomeController: Controller {
    private readonly IMusicManager _musicManager;
    public HomeController(IMusicManager musicManager) {
      _musicManager = musicManager;
    }
    public IActionResult Index() {
      var songs = _musicManager.GetAllMusic();
      return View(songs);
    }
  }
}
```

The preceding code first defines a private read-only field of the IMusicManager interface type. Making it read-only and private is considered the best practice, as this prevents you from accidentally assigning the field to a different value within your class. The next line of code defines the constructor class and uses the "constructor injection" approach to initialize the dependency object. In this case, any methods within the HomeController class will be able to access the _musicManager field and invoke all its available methods and properties. We'll talk more about the different types of DI later in this module.

The current code now supports the DI pattern since we are no longer passing concrete dependency to the Controller methods when the class is constructed. With the interface abstraction, we no longer need to create a new instance of the concrete class to directly reference the GetAllMusic() method. But instead, we reference the interface field to access the method. In other words, our method is now loosely coupled with the actual class implementation. This helps us to maintain our code more easily and perform unit tests conveniently.

Registering the service

Finally, let's register the interface mapping with the DI container. Go ahead and navigate to the Startup.cs file and then add the following code within the ConfigureServices() method:

```
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 MucisManager 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. We'll talk more about DI containers later in this module.

Benefits of DI

As you have learned from our previous example, DI entails many benefits that make your ASP.NET Core application easy to maintain and evolve. These benefits include the following:

- 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.

Reviewing types of dependency injection

There are a few options when it comes to implementing DI within your ASP.NET Core applications, and these include the following approaches:

- Constructor injection
- Method injection
- Property injection
- View injection

Let's talk about each type in detail in the coming sections.

Constructor injection

We've seen how we can implement constructor injection earlier in our music list example. But to recap, 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. In fact, when you create an ASP.NET Core MVC 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;
  }
}
```

In the preceding code, notice that the concept is very much similar to our previous example when we swapped out the MusicManager class reference with the IMusicManager interface to perform DI.

The ILogger<HomeController> 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.

Method injection

Method injection is another DI approach that allows you to inject lower-level dependent components as arguments into the method. In other words, dependent objects will be passed into the method instead of passing them into the class constructor. Implementing method injection is very helpful when various methods in your class need to invoke a child object dependency to complete their job. A typical example is writing to different log formats based on which methods are invoked. Let's take an actual example for you to better understand this approach.

Let's extend our previous example about the music list, but this time, we are going to implement something like a notifier to demonstrate method or function injection.

To start off, create a new interface called INotifier, as shown in the following code block:

```
namespace DI.DataManager {
  public interface INotifier {
    bool SendMessage(string message);
  }
}
```

In the preceding code, we have defined a simple interface that contains a single method called SendMessage. The method accepts a string parameter that represents a message, and returns a boolean type to determine whether the operation has succeeded or failed. It is as simple as that.

Now, let's proceed by creating a concrete class that implements the INotifier interface. Here's what the class declaration looks like:

```
namespace DI.DataManager {
  public class Notifier : INotifier {
    public bool SendMessage(string message) {
      //some logic here to publish the message
      return true;
    }
  }
}
```

The preceding code shows how the SendMessage() method is implemented. Notice that there's really no logic implemented within the method other than returning the boolean value of true. That was intentional because the implementation is irrelevant to this topic, and we don't want to draw your attention to that area. However, in real applications, you might create different classes to implement the logic for sending the message. For example, you could use message queues, pub/sub, Event Bus, email, SMS, or even a REST API call to broadcast the messages.

Now that we have our notifier object abstracted via an interface. Let's modify the IMusicManager interface to include a new method called GetAllMusicThenNotify. The updated IMusicManager.cs file should now look like this:

```
using DI.Models;
using System.Collections.Generic;
namespace DI.DataManager {
  public interface IMusicManager {
    List<SongModel> GetAllMusic();
    List<SongModel> GetAllMusicThenNotify(INotifier
    notifier);
  }
}
```

Notice that the GetAllMusicThenNotify() method also returns a List of SongModel objects, but this time, we are passing the INotifier interface as an argument.

Let's continue by implementing the GetAllMusicThenNotify() method within the MusicManager class.

Here's the code implementation of the method:

```
public List<SongModel> GetAllMusicThenNotify(INotifier notifier) {
  //invoke the notifier method
  var success = notifier.SendMessage("User viewed the music list page.");
  //return the response
  return success
          ? GetAllMusic()
          : Enumerable.Empty<SongModel>().ToList();
}
```

The preceding code invokes the SendMessage() method of the INotifier interface and then passes the message as the parameter/argument. This process is called method injection because we have injected the INotifier interface into the GetAllMusicThenNotify() method, hence, without having to instantiate the concrete implementation of the notifier object. Keep in mind that in this particular example, the SendMessage() method will always return true just to simulate the process and doesn't contain any actual implementation. This simply means that the value of the success variable will always be true.

The second line in the preceding code returns the response and uses the C# ternary conditional operator (?:) to evaluate what data the method should return based on the expression value. The Ternary operator is the simplified syntax of the if-else statement. In this case, we invoke the GetAllMusic() method to return the entire list of music if the value of the success variable is true, otherwise we return an empty list using the Enumerable.Empty<T> method. For more information about ternary operators and the Enumerable.Empty LINQ extension method, refer to https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.empty.

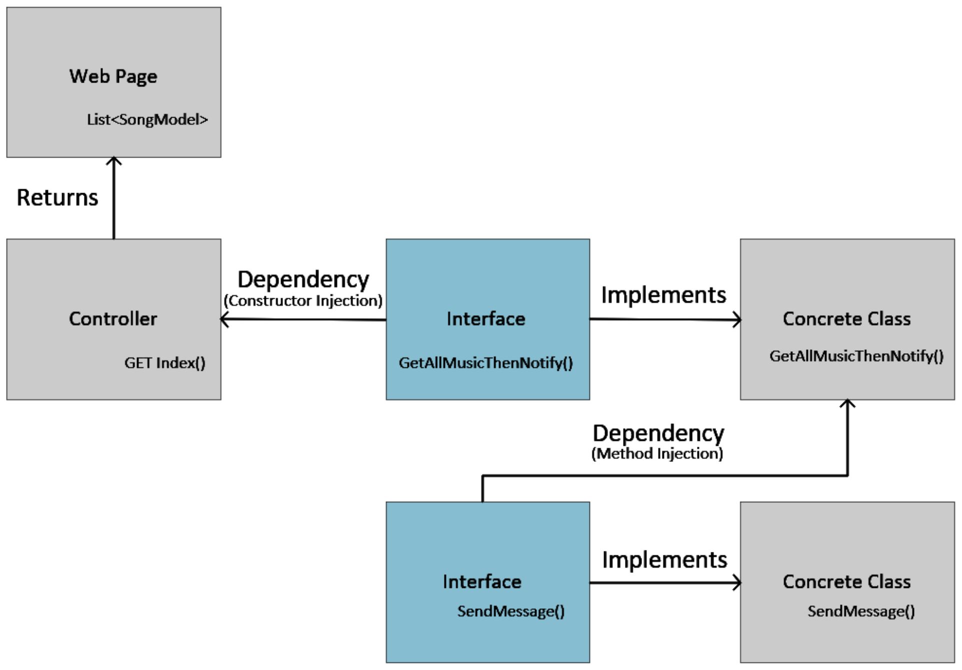
Now, the final step to perform is to update the Index() action method in the HomeController class to make use of the GetAllMusicThenNotify() method.

Here's the updated version of the method:

```
public IActionResult Index() {
  var songs = _musicManager.GetAllMusicThenNotify(new Notifier());
  return View(songs);
}
```

Notice in the preceding code that we are now passing the concrete instance of the notifier object. The GetAllMusicThenNotify() method will automatically resolve it because the concrete instance implements the INotifier interface.

To better understand how the dots connect to the picture, here's a high-level graphical representation of what we just did:



The important boxes in the preceding diagram are the Interface boxes. This is because abstracting your implementation via the interface enables you to avoid direct class access, and decouples various implementations in different classes. For example, if business requirements arise and ask you to implement different forms of notification based on different events, you could easily create SMSNotifier, MessageQueueNotifier, and EmailNotifier that implement the INotifier interface. Then, perform whatever

logic it requires to fulfill the business needs separately. While you may still be able to accomplish method injection without the use of an interface, chances are that it makes your code messy and very difficult to manage. Without using an interface, you would end up creating different methods for each of your notification classes, which leads you to back to unit tests and code maintenance issues.

Property injection

Property injection (or setter injection) allows you to reference a lower-level dependent component as a property in your class. You would only use this approach in case the dependency is truly optional. In other words, your service can still work properly without these dependencies provided.

Let's take another example using our existing music list sample. This time, we will update the Notifier sample to use property injection instead of method injection. The first thing that we need to do in order to make this happen is to update the IMusicManager interface. Go ahead and replace the existing code so that it will look similar to this:

```
using DI.Models;
using System.Collections.Generic;
namespace DI.DataManager {
  public interface IMusicManager {
    INotifier Notify { get; set; }
    List<SongModel> GetAllMusic();
    List<SongModel> GetAllMusicThenNotify();
  }
}
```

What we did in the preceding code is that we added a new property called Notify and then modified the GetAllMusicThenNotify() method by removing the INotifier parameter.

Next, let's update the MusicManager class to reflect the changes in the IMusicManager interface. The updated class should now look like this:

```
using System.Collections.Generic;
using System.Linq;
namespace DI.DataManager {
  public class MusicManager : IMusicManager {
    public INotifier Notify { get; set; };
    public List<SongModel> GetAllMusic() {
      //removed code for brevity
    }
    public List<SongModel> GetAllMusicThenNotify() {
      // Check if the Notify property has been set
      if (Notify != default) {
        //invoke the notifier method
        Notify.SendMessage("User viewed the music list
        page.");
      }
      //return list of music
      return GetAllMusic();
    }
  }
}
```

In the preceding code, we've implemented the Notify property, which returns an INotifier interface type using C#'s auto-implemented property feature. If you are not familiar with auto-properties, it basically makes property declaration more concise when no additional logic is required in the property accessors. What this means is that the following line of code:

```
public INotifier Notify { get; set; }
```

Is simply equivalent to the following code:

```
private INotifier _notifier;
public INotifier Notify {
  get { return _notifier };
  set { _notifier = value };
}
```

The preceding code can also be rewritten using Expression-Bodied Property Accessors, which was introduced in C# 7.0:

```
private INotifier _notifier;
public INotifier Notify {
  get => _notifier;
  set => _notifier = value;
}
```

You may use the preceding code when you need to set properties with different implementations. However, in the case of our example, using auto-properties makes more sense as it's cleaner.

Going back to our example, we need to implement the Notify property so that the HomeController class would be able to set its value before invoking the GetAllMusicThenNotify() method.

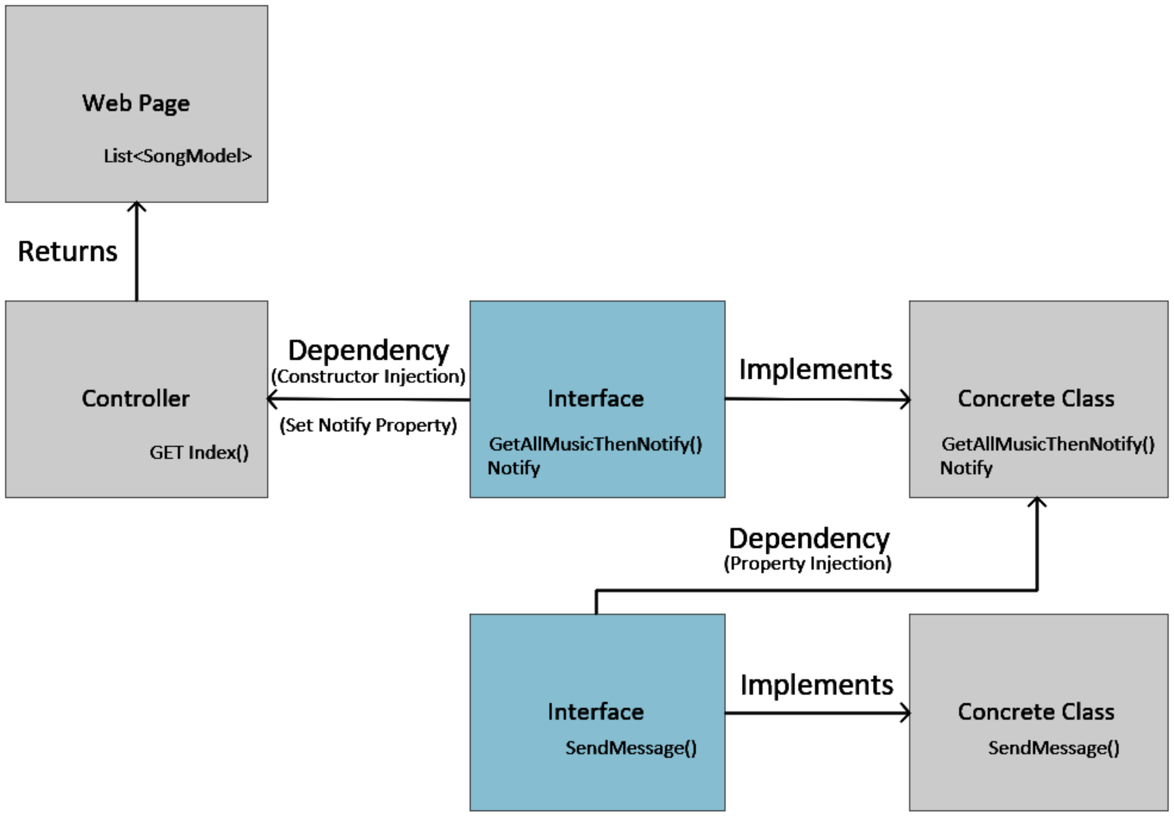
The GetAllMusicThenNotify() method is pretty much straightforward. First, it checks whether the Notify property has been set or is not null. The default keyword value of any reference type is null. In other words, validating against null or default doesn't matter here. Without the null validation check, you will end up getting a NullReferenceException error when the property is not set. So, it's a best practice to always check for nulls. Now, if the Notify property is not null, we then invoke the SendMessage() method. Finally, we return the list of music to the caller.

The final step that we need to modify is the Index() method of HomeController. Here's what the updated code looks like:

```
public IActionResult Index() {
  _musicManager.Notify = new Notifier();
  var songs = _musicManager.GetAllMusicThenNotify();
  return View(songs);
}
```

The preceding code sets the Notify property with a new instance of the Notifier class. It then invokes the GetAllMusicThenNotify() method and finally returns the result to the View.

Here's a high-level graphical representation of what we just did:



The important thing to note in this approach is that even if we don't set the Notify property, the Index() method will still work and returns the data to View. In summary, you should only use property injection when integrating optional features in your code.

View injection

View injection is another DI approach supported by ASP.NET Core. This feature was introduced in ASP.NET MVC 6, the first version of ASP.NET Core (previously known as ASP.NET 5), using the @inject directive. The @inject directive allows you to inject some method calls from a class or service directly into your View. This can be useful for view-specific services, such as localization or data required only for populating view elements.

Let's jump ahead with some examples. Now, add the following method within the MusicManager class:

```
public async Task<int> GetMusicCount() {
  return await Task.FromResult(GetAllMusic().Count);
}
```

The preceding code is an asynchronous method that returns a Task of int. While this book does not cover C# asynchronous programming in depth, perhaps providing a little bit of background about it is useful. The logic within the method simply returns the count of items from the GetAllMusic() result. The value of Count is obtained using the Count property of the List collection. Since the method expects a Task to be returned, and the GetAllMusic() method returns a List type, then the result is wrapped inside the Task.FromResult() call. It then uses the await operator to wait for the async method to complete the task, and then asynchronously returns the result to the caller when the process is complete. In other words, the await keyword is where things can get asynchronous. The async keyword enables the await keyword in that method and changes how method results are handled. In other words, the async keyword only enables the await keyword. For more information about C#'s async and await keywords, check out the reference links at the end of this module.

The next step that we need to perform in order for it to work is to register the MusicManager class as a service in the ConfigureServices() method of the Startup.cs file:

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

In the preceding code, we have registered the service as Transient. This means that every time the dependency is requested, a new instance of the service will be created. We'll talk more about service lifetimes in the Understanding dependency lifetimes section of this module.

Now, here's how you would inject the MusicManager class as a service in the View:

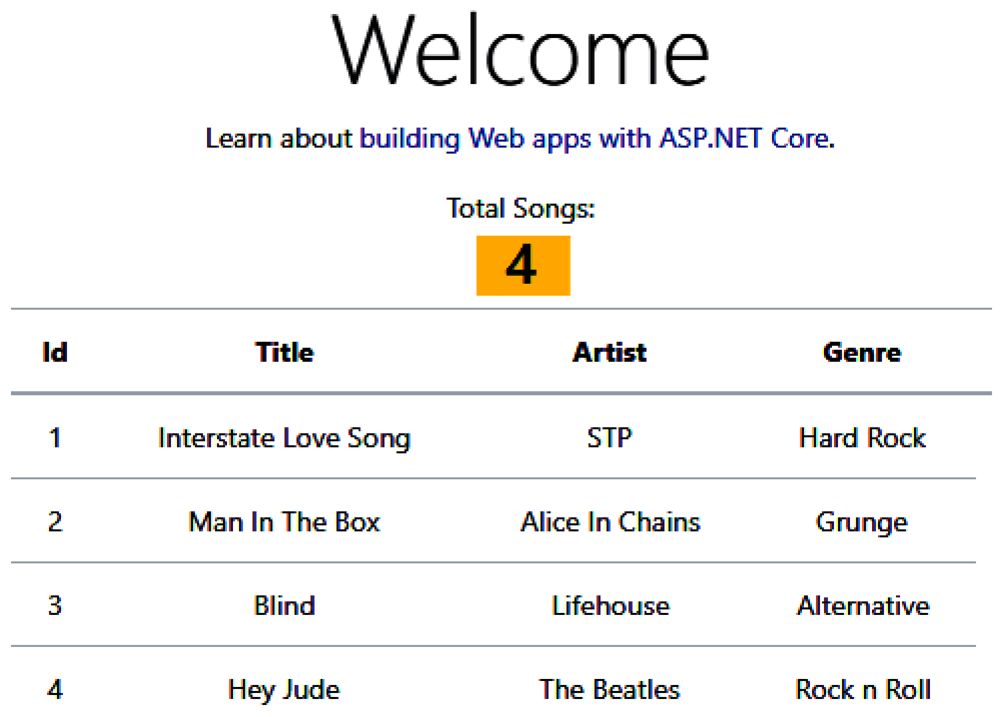
```
@inject DI.DataManager.MusicManager
MusicService
```

And here's the code for referencing the GetMusicCount() method that we added earlier:

```
Total Songs: <h2>@await MusicService.GetMusicCount()</h2>
```

The @ symbol is a Razor implicit syntax that allows you to use C# code in the View. We'll deep dive into Razor in the next module.

Here is a sample screenshot of the output after a service has been injected into the View:



Welcome

Learn about building Web apps with ASP.NET Core.

Total Songs:

4

Id	Title	Artist	Genre
1	Interstate Love Song	STP	Hard Rock
2	Man In The Box	Alice In Chains	Grunge
3	Blind	Lifehouse	Alternative
4	Hey Jude	The Beatles	Rock n Roll

Notice that the value of 4 has been printed on the page. That's the value returned from the GetMusicCount() method. Keep in mind that while using this technique might be useful, you should consider separating your View and Controller logic to value the separation of concerns. In practice, it's recommended to generate the data from your Controller; the View should not care how and where the data was processed.

Now that we've seen the different types of DI and learned when to use them, we'll move on to discuss DI containers in the next section.

Understanding dependency injection containers

The **dependency injection container** is not really a requirement to apply the DI technique. However, using it can simplify the management of all of your dependencies, including their lifetimes, as your application grows and becomes more complex.

.NET Core comes with a built-in DI/IoC container that simplifies DI management. In fact, the default ASP.NET Core application template uses DI extensively. You can see it by looking at the Startup class of your ASP.NET Core application:

```
public class Startup {
  public IConfiguration Configuration { get; }
  public Startup(IConfiguration configuration) {
    Configuration = configuration;
  }
  public void ConfigureServices(IServiceCollection services) {
    // This method gets called by the runtime.
    // Use this method to add services to the container.
  }
  public void Configure(IApplicationBuilder app,
  IWebHostEnvironment env) {
    // This method gets called by the runtime.
    // Use this method to configure the HTTP request
    // and middleware pipeline.
  }
}
```

In the preceding code, the IConfiguration interface has been passed to the Startup class constructor using the constructor injection approach. This allows you to get access to the configuration values defined in the appsettings.json file. You don't need to register IConfiguration yourself as the framework takes care of this for you when the Host is configured. You can see how this is being done by looking at the CreateHostBuilder() method of the Program class:

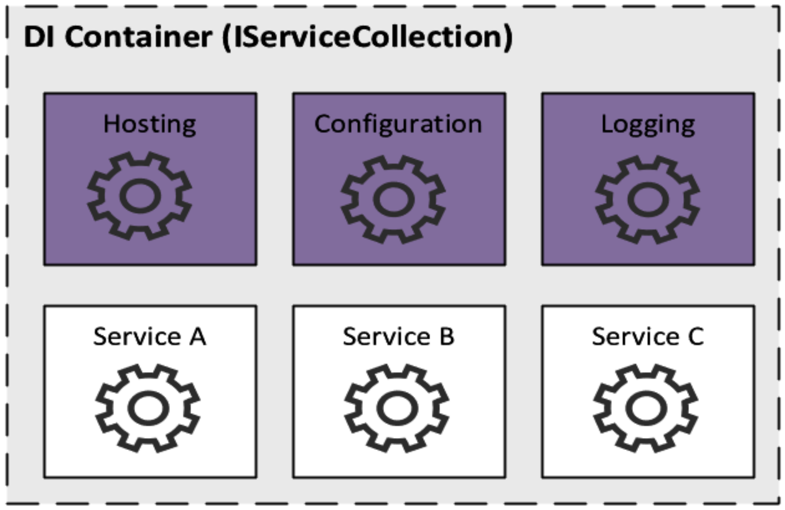
```
public static IHostBuilder CreateHostBuilder(string[] args) =>
       Host.CreateDefaultBuilder(args)
       .ConfigureWebHostDefaults(webBuilder => {
       webBuilder.UseStartup<Startup>();
       }

);
```

The CreateDefaultBuilder() method in the preceding code initializes a new instance of the WebHostBuilder class with pre-configured defaults, including Hosting, Configurations, and Logging. Ultimately, the ConfigureWebHostDefaults() method adds everything else needed for a typical ASP.NET Core application, such as configuring Kestrel and using the Startup class to configure your DI container and middleware pipeline.

Keep in mind that you can only inject certain services into the Startup class constructor, and these include IWebHostEnvironment, IhostEnvironment, and IConfiguration.

Other services must be registered to the DI container when the application starts. This process is done by adding services to IServiceCollection:



In .NET Core, the dependencies managed by the container are called services. Any services that we expect to be injected into the container must be added to IServiceCollection so that the service provider will be able to resolve the services at runtime. Under the hood, the Microsoft built-in DI container implements the IServiceProvider interface. It's really not ideal to build your own IoC/DI container framework, but if you do, the IServiceProvider interface is what you should look at.

IServiceCollection has two main types of services:

- **Framework-provided services:** These represent the purple boxes from the preceding diagram, which are part of the .NET Core framework and registered by default. These services include Hosting, Configuration, Logging, HttpContext, and many others.
- **Application services:** These represent the white boxes. This type of services refers to the services that you create and use in your ASP.NET Core application that is not part of the framework itself. Since these services are typically created by you, then you need to manually register them in the DI container so that they will be resolved when the application starts. An example of this type of service is our IMusicManager interface sample.

The DI container manages the instantiation and configuration of the services registered. Typically, this process is executed in three steps:

1. **Registration:** The services that you want to be injected into different areas of your application need to be registered first so that the DI container framework will know which implementation type to map the service to. A great example of this is when we mapped the IMusicManager interface to the concrete class implementation called MusicManager. Generally, service registrations are configured in the ConfigureServices() method of the Startup.cs file, as in the following code:

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

2. **Resolution:** This is where the DI container automatically resolves the dependency when the application starts by creating an object instance and injecting it into the class. Based on our previous example, this is where we inject the IMusicManager interface into the HomeController class constructor using the constructor injection approach, as shown in the following code:

```
private readonly IMusicManager _musicManager;
public HomeController(IMusicManager musicManager) {
  _musicManager = musicManager;
}
```

3. **Disposition:** When registering services, the DI container framework also needs to know the lifetime of the dependencies so it can manage them correctly. Based on our previous example regarding the constructor injection approach, this is where we register the interface mapping as a Transient service in the ConfigureServices() method of the Startup.cs file.

For more information about the ASP.NET Core fundamentals and how the default Microsoft DI container works under the hood, refer to the official documentation here: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection.

Understanding dependency lifetimes

If you're completely new to ASP.NET Core, or haven't worked with ASP.NET Core for a long time, or if you're an experienced ASP.NET developer but don't really look into dependency lifetimes in detail, the chances are you might be using just one type of dependency lifetime to register all your services when building ASP.NET Core applications. This is because you are confused as to which service lifetime to use, and you wanted to play it safe. Well, that's understandable, because choosing which type of service lifetime to use can be confusing sometimes. Hopefully, this section will give you a better understanding of the different types of lifetimes that you can use within your application and decide when to use each option.

There are primarily three service lifetimes in ASP.NET Core DI:

- Transient
- Scoped
- Singleton

Transient service

The AddTransient() method is probably what you were using most often. If that is the case, then that's a good call because this type is the safest option to use when in doubt. Transient services are created each time they are requested. In other words, if you register your service with a transient lifetime, you will get a new object whenever you invoke it as a dependency, regardless of whether it is a new request or the same. This lifetime works best for lightweight and stateless services as they are disposed at the end of the request.

Let's take a look at an example for you to better understand how transient service lifetime works. We'll use the existing music list example for ease of reference. The first thing we need to do is add the following property to the IMusicManager interface:

```
Guid RequestId { get; set; }
```

The preceding code is just a simple property that returns a Globally Unique Identifier (GUID). We'll use this property to determine how each dependency behaves.

Now, let's implement the RequestId property in the MusicManager class by adding the following code to the existing code:

```
public Guid RequestId { get; set; }
public MusicManager(): this(Guid.NewGuid()) { }
public MusicManager(Guid requestId) {
  RequestId = requestId;
}
```

In the preceding code, we've implemented the RequestId property from the IMusicManager interface and then defined two new constructors. The first constructor sets a new GUID value, and the second constructor initializes the GUID value to the RequestId property by applying the constructor injection approach. Without the first constructor, the DI container won't be able to resolve the dependency that we've configured in the HomeController class when the application starts.

To demonstrate multiple dependency references, let's create a new class called InstrumentalMusicManager and then copy the following code:

```
using System;
namespace DI.DataManager {
  public class InstrumentalMusicManager {
    private readonly IMusicManager _musicManager;
    public Guid RequestId { get; set; }
    public InstrumentalMusicManager(IMusicManager
    musicManager) {
      RequestId = musicManager.RequestId;
    }
  }
}
```

In the preceding code, we've also applied the Constructor Injection approach by injecting the IMusicManager interface as an object dependency into the class. We then initialized the value of the RequestId property, just like what we did in the MusicManager class. The only differences between the InstrumentalMusicManager and MusicManager classes are the following:

1. The InstrumentalMusicManager class doesn't implement the IMusicManager interface. This was intentional because we are only interested in the RequestId property and to make this demo as simple as possible.
2. The InstrumentalMusicManager class doesn't have a setter constructor.The reason for this is that we will let the MusicManager class set the value. By injecting the IMusicManager interface into the constructor, we will be able to reference the value of the RequestId property from it since the MusicManager class implements this interface, although the value of the property will vary depending on how the service is registered with the type of lifetime, which we will see in action later.

Now, navigate to the Startup class and update the ConfigureServices() method so that it will look similar to the following code:

```
public void ConfigureServices(IServiceCollection services) {
  services.AddTransient<IMusicManager, MusicManager>();
  services.AddTransient<InstrumentalMusicManager>();
  // Removed for brevity. Register other services here
}
```

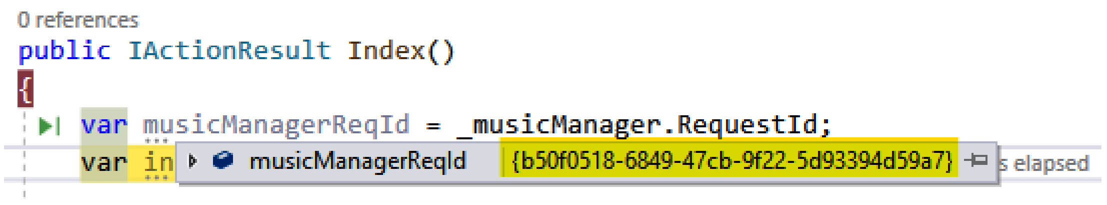
In the preceding code, we've registered both services as transient services. Notice that we opted out of the second parameter of the AddTransient() method. This is because the InstrumentalMusicManager class doesn't implement any interface.

The final step that we need to perform is to update the HomeController class to inject the InstrumentalMusicManager concrete class as a dependency and reference both RequestId values from each service that we have registered earlier. Here's what the HomeController class code looks like:

```
public class HomeController : Controller {
  private readonly IMusicManager _musicManager;
  private readonly InstrumentalMusicManager _insMusicManager;
  public HomeController(IMusicManager musicManager,
  InstrumentalMusicManager
  insMusicManager) {
    _musicManager = musicManager;
    _insMusicManager = insMusicManager;
  }
  public IActionResult Index() {
    var musicManagerReqId = _musicManager.RequestId;
    var insMusicManagerReqId = _insMusicManager.RequestId;
    _musicManager.Notify = new Notifier();
    var songs = _musicManager.GetAllMusicThenNotify();
    return View(songs);
  }
}
```

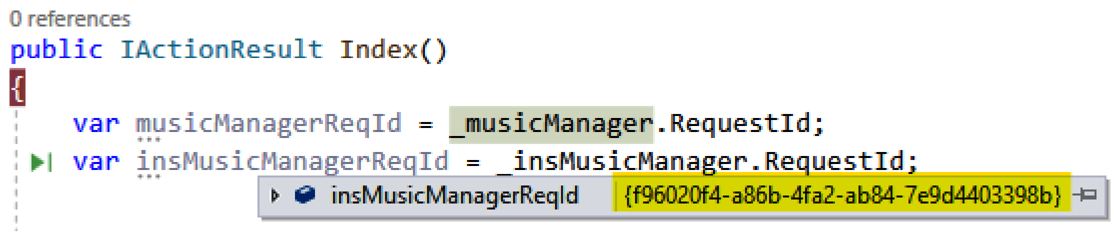
In the preceding code, we injected an instance of the InstrumentalMusicManager class and IMusicManager interface as a dependency using the Constructor Injection approach. We then get each RequestId value from both object instances.

Now, when you run the application and set a break point at the Index() method, we should see the different values for the musicManagerReqId and insMusicManagerReqId variables, as shown in the following screenshot:



```
0 references
public IActionResult Index()
{
    var musicManagerReqId = _musicManager.RequestId;
    var in  musicManagerReqId  {b50f0518-6849-47cb-9f22-5d93394d59a7}   s elapsed
```

In the preceding screenshot, we can see that the musicManagerReqId variable holds the GUID value of b50f0518-8649-47cb-9f22-59d3394d59a7. Let's take a look at the value of insMusicManagerReqId in the following screenshot:



```
0 references
public IActionResult Index()
{
    var musicManagerReqId = _musicManager.RequestId;
    var insMusicManagerReqId = _insMusicManager.RequestId;
          insMusicManagerReqId  {f96020f4-a86b-4fa2-ab84-7e9d4403398b}
```

As you can see, each variable has different values, even if the RequestId has only been set in the MusicManager class implementation. This is how the Transient services work, and the DI container framework creates a new instance for every dependency each time they are requested. This ensures the uniqueness of each dependent object instance for every request. While this service lifetime has its own benefits, be aware that using this type of lifetime can potentially impact the performance of your application, especially if you are working on a huge monolith app where dependency reference is massive and complex.

Scoped service

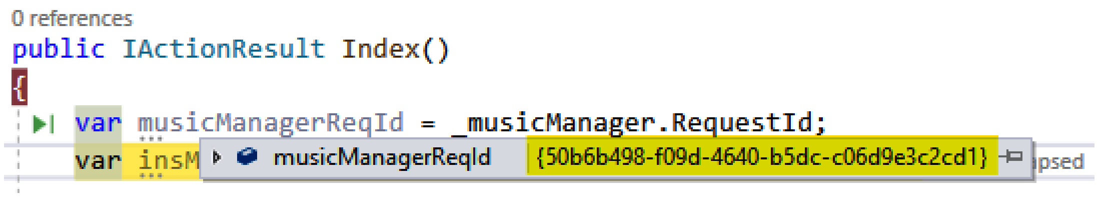
Scoped service lifetimes are services created at the lifetime of each client request. In other words, an instance is created per web request. A common example of using a Scoped lifetime is when using an Object Relational Mapper (ORM) such as Microsoft's Entity Framework Core (EF). By default, the DbContext in EF will be created once per client web request. This is to ensure that related calls to process the data will be contained in the same object instance for each request. Let's take a look at how this approach works by modifying our existing previous example.

Let's go ahead and update the ConfigureServices() method of the Startup class so that it will look similar to the following code:

```
public void ConfigureServices(IServiceCollection services) {
  services.AddScoped<IMusicManager, MusicManager>();
  services.AddTransient<InstrumentalMusicManager>();
}
```

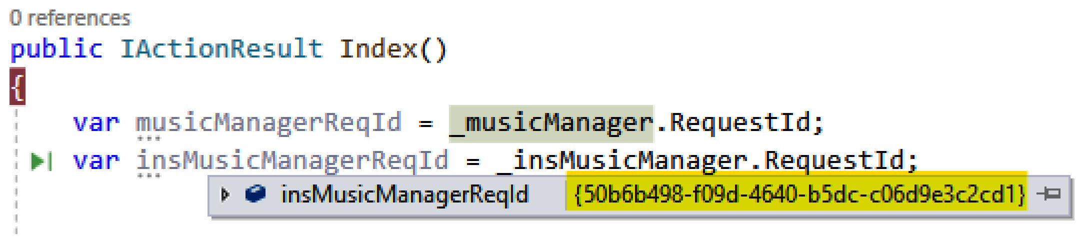
All that we actually changed in the preceding code is just the MusicManager class registration being added as a scoped service. The InstrumentalMusicManager interface remains transient because this class depends on the MusicManager class, which implemented the IMusicManager interface. This means that the DI container will automatically apply whatever service lifetime is being used in the main component.

Now, when you run the application again, you should see that both the musicManagerReqId and insMusicManagerReqId variables now hold the same RequestId value, as shown in the following screenshot:



```
0 references
public IActionResult Index()
{
    var musicManagerReqId = _musicManager.RequestId;
    var insM  musicManagerReqId  {50b6b498-f09d-4640-b5dc-c06d9e3c2cd1}  psed
```

In the preceding screenshot, we can see that the musicManagerReqId variable holds the GUID value of 50b6b498-f09d-4640-b5dc-c06d9e3c2cd1. The value of the insMusicManagerReqId variable is shown in the following screenshot:



```
0 references
public IActionResult Index()
{
    var musicManagerReqId = _musicManager.RequestId;
    var insMusicManagerReqId = _insMusicManager.RequestId;
            insMusicManagerReqId  {50b6b498-f09d-4640-b5dc-c06d9e3c2cd1}
```

Notice in the preceding screenshot that both musicManagerReqId and insMusicManagerReqId now have the same value. This is how Scoped services work; the values will remain the same throughout the entire client request.

Singleton service

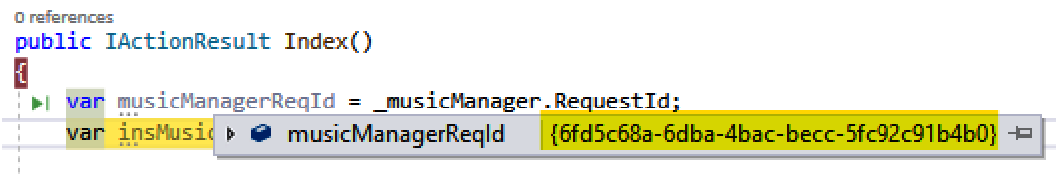
Singleton service lifetimes are services created only once and all dependencies will share the same instance of the same object during the entire lifetime of the application. You would use this type of lifetime for services that are expensive to instantiate because objects will be stored in memory and can be reused for all injections within your application.

A typical example of a singleton service is ILogger. The ILogger<T> instances for a certain type, T, are kept around for as long as the application is running. What this means is that when injecting an ILogger<HomeController> instance into your Controller, the same logger instance will be passed to it every time.

Let's take a look at another example to better understand this type of service lifetime. Let's update the ConfigureServices() method in the Startup class and add MusicManager as a singleton service, just as in the following code:

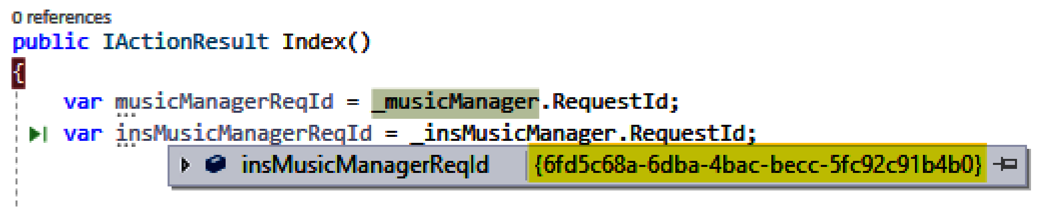
```
public void ConfigureServices(IServiceCollection services) {
  services.AddSingleton<IMusicManager, MusicManager>();
  services.AddTransient<InstrumentalMusicManager>();
}
```

The AddSingleton() method in the preceding code enables the service to be created only once. When we run the application again, we should be able to see that both the musicManagerReqId and insMusicManagerReqId variables now hold the same RequestId value, as shown in the following screenshots:



```
0 references
public IActionResult Index()
{
    var musicManagerReqId = _musicManager.RequestId;
    var insMusic  musicManagerReqId  {6fd5c68a-6dba-4bac-becc-5fc92c91b4b0}
```

In the preceding screenshot, we can see that the musicManagerReqId variable holds the GUID value of 6fd5c68a-6dba-4bac-becc-5fc92c91b4b0. Now, let's take a look at the value of the insMusicManagerReqId variable in the following screenshot:



```
0 references
public IActionResult Index()
{
    var musicManagerReqId = _musicManager.RequestId;
    var insMusicManagerReqId = _insMusicManager.RequestId;
            insMusicManagerReqId  {6fd5c68a-6dba-4bac-becc-5fc92c91b4b0}
```

As you notice in the preceding screenshot, the value of each variable is also the same. The only difference to this approach compared with Scoped services is that no matter how many times you make a request to the Index() action method, you should still be getting the same value. You can verify this by refreshing the page to simulate multiple HTTP requests. In the web context, this means that every subsequent request will use the same object instance as it was first created. This also means that it spans across web requests, so regardless of which users made the request, they will still be getting the same instance.

Keep in mind that since singleton instances are kept in memory during the entire application's lifetime, you should watch out for your application memory usage. The good thing though is that the memory will be allocated just once, so the garbage collector will have less to do and may provide you with some performance gain. However, I would recommend that you only use a singleton when it makes sense and don't make things a singleton because you think it's going to save on performance. Moreover, don't mix a singleton service with other service lifetime types, such as transient or scoped, because it may affect how complex scenarios your application behaves.

For more advance and complex scenarios, visit the official documentation relating to DI in ASP.NET Core at https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection.

Learning and understanding how each service lifetime works is very important in order for your application to behave correctly. Now, let's take a quick look at how we can manage services for handling complex scenarios in the next section.

Handling complex scenarios

If you've made it this far, then we can assume that you now have a better understanding of how the DI works and how you could implement them in different scenarios as required. In this section, we are going to look at some complex situations that you might face when writing your applications. We will see how we can apply the available options provided by the default DI containers to solve complex scenarios. Finally, we are going to look at how we can improve the organization of services when registering them in the DI container.

Service descriptors

It's important to understand what service descriptors are before we dive into various complex scenarios.

Service descriptors contain information about the registered services that have been registered in the DI container, including the type of service, implementation, and lifetime. These are used internally by both IServiceCollection and IServiceProvider. It's very uncommon for us to work directly against service descriptors since they are typically created automatically by the various extension methods of IServiceCollection. However, situations may arise that may require you to work directly with service descriptors.

Let's take a look at some examples to make sense of this. In our previous example, we've registered the IMusicManager interface mapping as a service using the AddSingleton() generic extension method:

```
services.AddSingleton<IMusicManager, MusicManager>();
```

Using the generic extension method in the preceding code is very convenient to use when registering our services in the DI container. However, there may be scenarios where you would want to add services manually using service descriptors. Let's see how we can achieve this by looking at some examples.

There are four possible ways to create service descriptors. The first one is to use the ServiceDescriptor object itself, and pass the required arguments in the constructor, as shown in the following code snippet:

```
var serviceDescriptor = new ServiceDescriptor
(
typeof(IMusicManager),
typeof(MusicManager),
ServiceLifetime.Singleton
);

services.Add(serviceDescriptor);
```

In the preceding code, we've passed IMusicManager in the first argument as the service type. We then set the corresponding implementation type as MusicManager and finally, set the service lifetime to a singleton. The ServiceDescriptor object has another two overload constructors that you can use. You can read more about them at https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.servicedescriptor.

The second option is to use the static Describe() method of the ServiceDescriptor object, as shown in the following code snippet:

```
var serviceDescriptor = ServiceDescriptor.Describe
(
typeof(IMusicManager),
typeof(MusicManager),
ServiceLifetime.Singleton
);
services.Add(serviceDescriptor);
```

In the preceding code, we are passing the same arguments to the method, which is pretty much the same as what we did earlier using the ServiceDescriptor object constructor option. You can read more about the Describe() method and its available overload methods at https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.servicedescriptor.describe.

You may have noticed that both options in the preceding examples require us to pass the service lifetime. In this case, we are forced to pass the ServiceLifetime.Singleton enum value. To simplify them, we can use the available static methods to create service descriptors with lifetimes.

The following code demonstrates the remaining options:

```
var serviceDescriptor = ServiceDescriptor.Singleton
(
typeof(IMusicManager),
typeof(MusicManager)
);

services.Add(serviceDescriptor);
```

The preceding code makes use of the Singleton() static method by simply passing both the service type and the corresponding implementation type. While the code seems much cleaner now, you can simplify the creation further by using the generic method to make your code more concise, as shown in the following code snippet:

```
var serviceDescriptor = ServiceDescriptor
       .Singleton<IMusicManager, MusicManager>();
services.Add(serviceDescriptor);
```

Add versus TryAdd

We've learned how to create service descriptors in the previous example. In this section, let's take a look at the various ways in which we can register them in the DI container.

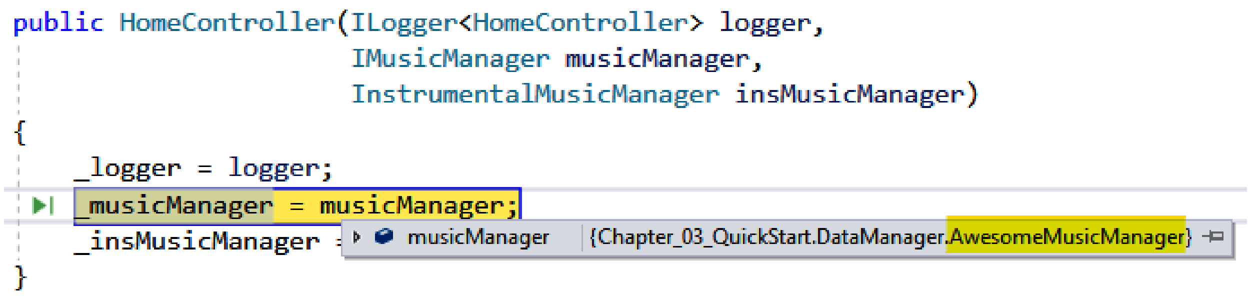
Earlier in this module, we've seen how to use the generic Add extension methods, such as the AddTransient, AddScoped, and AddSingleton methods for registering a service in the DI container with a specified lifetime. Each of these methods has various overloads that accept different arguments based on your needs. However, as your application becomes more complex and you have a lot of services to deal with, using these generic methods can potentially cause your application to behave differently when you accidentally register the same type of service.

For example, register the following service multiple times:

```
services.AddSingleton<IMusicManager, MusicManager>();
services.AddSingleton<IMusicManager, AwesomeMusicManager>();
```

The preceding code registers two services that refer to the IMusicManager interface. The first registration maps to the MusicManager concrete class implementation, and the second one maps to the AwesomeMusicManager class.

If you run the application, you will see that the implementation type being injected intothe HomeController class is the AwesomeMusicManager class, as shown in the following screenshot:



```
public HomeController(ILogger<HomeController> logger,
                      IMusicManager musicManager,
                      InstrumentalMusicManager insMusicManager)
{
    _logger = logger;
    _musicManager = musicManager;
    _insMusicManager =
}
```

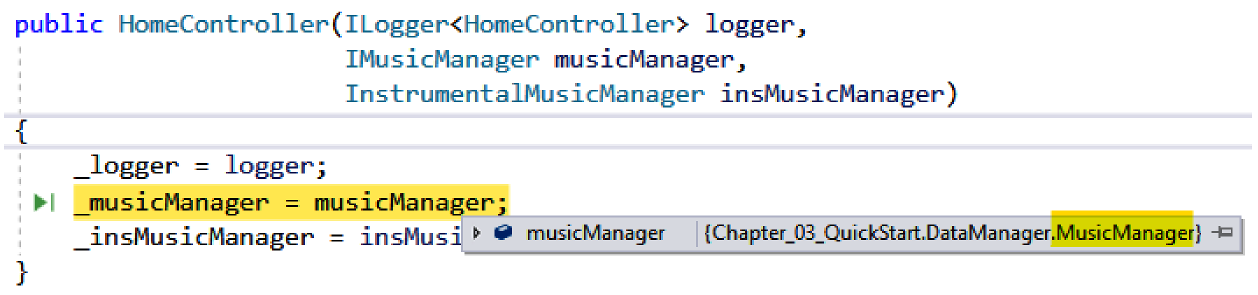
```
musicManager    {Chapter_03_QuickStart.DataManager.AwesomeMusicManager}
```

This simply means that the DI container will use the last registered entry for situations where you register multiple services of the same type. Therefore, the order of service registrations in the ConfigureServices() method can be quite important. To avoid this kind of situation, we can use the various TryAdd() generic extension methods that are available for registering the service.

So, if you want to register multiple implementations of the same service, you can simply do something like this:

```
services.AddSingleton<IMusicManager, MusicManager>();
services.TryAddSingleton<IMusicManager, AwesomeMusicManager>();
```

In the preceding code, we've changed the second registration to make use of the TryAddSingleton() method. When you run the application again, you should now see that the MusicManager class implementation is the one that gets injected as shown in the following figure:



```
public HomeController(ILogger<HomeController> logger,
                      IMusicManager musicManager,
                      InstrumentalMusicManager insMusicManager)
{
    _logger = logger;
    _musicManager = musicManager;
    _insMusicManager = insMusi
}
```

```
musicManager    {Chapter_03_QuickStart.DataManager.MusicManager}
```

When using TryAdd() methods, the DI container will only register services when there is no implementation already defined for a given service type. This makes things convenient for you, especially when you have complicated applications, because you can express your intent more clearly when registering your service and it prevents you from accidentally replacing previously registered services. So, if you want to register your services safely, then consider using the TryAdd() method instead.

Dealing with multiple service implementations

Previously, we've seen the effect of using the Add() methods for registering multiple services of the same service type with the DI container. While the DI container uses the last implementation type defined for the same service type, you should know that the first service defined is still kept in the service collections entry. In other words, invoking the Add() method multiple times for the same interface will create multiple entries in the service collection. This means that the last registration in our previous example does not replace the first registration.

To utilize the multiple implementations of the same interface, then you must first change how you define your services with having the same service type. This is to avoid potential side effects when having duplicate instances of the implementation. Therefore, when registering multiple instances of an interface, it's recommended to use the TryAddEnumerable() extension method, just as in the following example:

```
services.TryAddEnumerable(
       ServiceDescriptor.Singleton<IMusicManager,MusicManager>());
services.TryAddEnumerable(
       ServiceDescriptor.Singleton<IMusicManager,AwesomeMusicManager>());
```

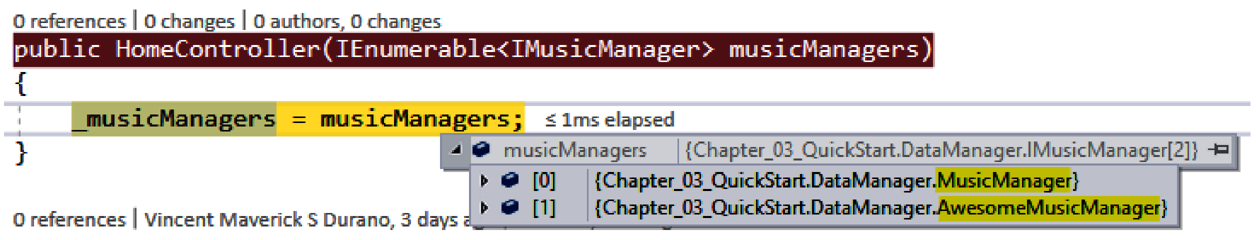
In the preceding code, we've replaced the AddSingleton() and TryAddSingleton() calls to the TryAddEnumerable() method. The TryAddEnumerable() method accepts a ServiceDescriptor argument type. This method prevents duplicate registrations of the same implementation. For more information, see https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.extensions.servicecollectiondescriptorextensions.tryaddenumerable

Now, the next step is to modify the HomeController class and contain the dependencies in an IEnumerable generic collection type to allow all implementations to be evaluated and resolved.

Here's an example of how to do that using our previous example:

```
private readonly IEnumerable<IMusicManager> _musicManagers;
public HomeController(IEnumerable<IMusicManager> musicManagers) {
  _musicManagers = musicManagers;
}
```

In the preceding code, we've changed the HomeController constructor argument to accept an IEnumerable<IMusicManager> service type. When the DI container is resolving services for this class, it will now attempt to resolve all instances of IMusicManager and inject them as an IEnumerable, as shown in the following screenshot:



```
0 references | 0 changes | 0 authors, 0 changes
public HomeController(IEnumerable<IMusicManager> musicManagers)
{
    _musicManagers = musicManagers;  ≤ 1ms elapsed
}
0 references | Vincent Maverick S Durano, 3 days a
```

```
musicManagers   {Chapter_03_QuickStart.DataManager.IMusicManager[2]}
  [0]  {Chapter_03_QuickStart.DataManager.MusicManager}
  [1]  {Chapter_03_QuickStart.DataManager.AwesomeMusicManager}
```

Keep in mind that the DI container will only resolve multiple instances of service implementations when the type is IEnumerable.

Replacing and removing service registrations

In this section, we'll take a look at how we can replace and remove service registrations. To replace a service registration, you can use the Replace() extension method of the IServiceCollection interface, as shown:

```
services.AddSingleton<IMusicManager, MusicManager>();
services.Replace(ServiceDescriptor.Singleton<IMusicManager, AwesomeMusicManager>());
```

The Replace() method also accepts a ServiceDescriptor argument type. This method will look for the first service registration for the IMusicManager service type and then remove it if it finds one. The new implementation type will then be used to create a new registration in the DI container. In this case, the MusicManager implementation type will be replaced with the AwesomeMusicManager class implementation. One thing to keep in mind here is that the Replace() method will only support removing the first service type entry in the collection.

In situations where you would need to remove all prior service registrations of a service type, you can use the RemoveAll() extension method and pass the type of the service that you wish to remove. Here's an example:

```
services.AddSingleton<IMusicManager, MusicManager>();
services.AddSingleton<IMusicManager, AwesomeMusicManager>();
services.RemoveAll<IMusicManager>();
```

The preceding code removes both registrations of the IMusicManager service type in the service collection.

Replacing or removing services in the DI container is quite a rare scenario, but it may be useful if you want to provide your own implementation for the framework or other third-party services.

Summary

DI is a huge topic, but we've tackled most of the major topics that should help you as a beginner as you progress on your journey to learning ASP.NET Core.

We've covered the concepts of DI, how it works under the hood, and its basic usage in the context of ASP.NET Core. These concepts are crucial for creating well-designed and well-decoupled applications. We've learned that DI offers a few benefits that help us to build robust and powerful applications. By following some detailed examples, we've learned how we can effectively use DI to solve potential problems in a variety of scenarios.

DI is a very powerful technique for building highly extensible and maintainable applications. By taking advantage of the abstractions, we can easily swap out dependencies without affecting the behavior of your code. This gives you greater flexibility in terms of integrating new features easily, and makes your code more testable, which is also crucial for building well-crafted applications. While the DI container is not really a requirement to apply the DI pattern, using it can simplify the management of all of your dependencies, including their lifetimes, as your application grows and becomes more complex.

In the next module, we are going to explore Razor View Engines for building powerful ASP.NET Core web applications. We will do some hands-on coding by building the application from scratch so that you have a better understanding of the topics as you progress.