Course Overview

Course Overview

Hi everyone. My name is Steve Gordon, and welcome to my course, Dependency Injection in ASP.NET Core. I'm a Microsoft MVP, a senior developer working at Madgex Limited, and a passionate member of the .NET community. ASP.NET Core is a blazingly fast web application framework that has been built on the foundation of dependency injection. This course will get you fully up to speed on dependency injection and its use in ASP.NET Core applications. You'll be able to apply these skills to build powerful, real-world ASP.NET Core applications. Some of the major topics that we'll cover include exploring the features provided by the Microsoft dependency injection container, learning how to correctly register services with the dependency injection container, learning how to resolve services via dependency injection across your ASP.NET Core application, and learning how to extend and replace the built-in container when faced with more complex application requirements. By the end of this course, you'll have a complete understanding of the capabilities of the Microsoft dependency injection container. You'll be able to apply this knowledge in your ASP.NET Core applications to take full advantage of the benefits of dependency injection. Before beginning the course, you should be broadly familiar with general ASP.NET Core concepts and have some experience of building ASP.NET applications. I hope you'll join me on this journey to learn about the Microsoft dependency injection container with the Dependency Injection in ASP.NET Core course at Pluralsight.

Registering Your First Service

Introduction and Overview

Hello, and welcome to Dependency Injection in ASP.NET Core. I'm Steve Gordon, and in this course, I'm going to show you how to take full advantage of the built-in dependency injection container in your ASP.NET Core applications. I'm using the ASP.NET Core 2.2 release in this course. However, the dependency injection feature has changed very little since the original ASP.NET Core 1.0 release in June 2016. So most of the content we'll cover is applicable when using previous versions. In this module, we'll dive right in with an example of how using dependency injection can help us improve an application. We'll begin by looking at some design problems in the code for an existing application. We'll get our feet wet by refactoring the code to depend on abstractions and support dependency injection. We'll then make use of the built-in ASP.NET Core dependency injection container to register our first service. Throughout this module, we'll be refactoring an ASP.NET Core MVC application so that we can take advantage of dependency injection. If you'd like to follow along, I've included a before solution in the course downloads for this module. The solution relies on .NET Core 2.2, so you'll need the latest 2.2 SDK installed to build the solution. You can download this from the Microsoft website, dot.net. I'll be using Visual Studio 2017 to develop the solution, but you're free to use whichever IDE or editor you prefer. I'm really excited to teach you how to take full advantage of dependency injection in your ASP.NET Core applications. So with that, let's get started.

Why Use Dependency Injection?

Dependency injection, often abbreviated as simply DI, is a design pattern that supports development of loosely coupled code. The use of dependency injection supports the implementation of two related concepts, inversion of control and the dependency inversion principle. Dependency injection is a foundational concept in ASP.NET Core upon which the rest of the framework relies heavily. It is strongly encouraged that developers take advantage of dependency injection when building ASP.NET Core applications. In fact, you have to work quite hard against the framework in order to avoid using it. We'll begin by identifying some architectural problems in the before application. This page is the home page for a fictional tennis club. At the top of the home page, the owners have a requirement to display the current weather conditions near the club. Since I'm in the UK, perhaps I could've just hard-coded this as a rainy forecast. This is the ASP.NET Core controller, which handles request for the home page. It has an Index action method, which creates a simple viewModel that will be returned as the view result. This method makes use of a class called WeatherForecaster to get the current weather conditions. WeatherForecaster is a supporting class defined at a lower level in the code base of my application. In order to fully render the home page, the Index method makes use of this class. This is known as a dependency because the Index method is now dependent on the WeatherForecaster in order to complete its job. Currently, the Index method is responsible for the creation of the WeatherForecaster instance on which it depends. It does this using the new keyword. Whilst everything works as expected, there are a few problems with this current approach. Firstly, because it is responsible for creating an instance of it, the controller is now tightly coupled to the current WeatherForecaster implementation. Imagine that at some point in the future, I want to add an improved weather forecasting implementation to my application. Let's call it AmazingWeatherForecaster. If I then want the HomeController to use this improved implementation, I have to change the Index method so that it creates an instance of the AmazingWeatherForecaster instead of the original WeatherForecaster. This makes my application harder to maintain as changes can ripple up through the layers of my code base. I'll undo that change for now, and we'll continue using the original WeatherForecaster class. A second problem arises when I want to unit test the controller. The Index method has logic for adding a WeatherDescription to the viewModel for the page based on the weather result from the WeatherForecaster. It would, therefore, be reasonable to test that the expected WeatherDescription is added to the viewModel for each of the possible weather results that the WeatherForecaster may return. I've added two unit tests to cover this. The tests create an instance of the HomeController, my subject under test, and then invoke the Index method before asserting on the response. I'll run the tests, and we'll take a look at the results. One unit test has failed. My second test expects to test the case where the rain condition is returned. However, since the Index action has the responsibility for creating the WeatherForecaster dependency directly, when under test, I can't control the response that the GetCurrentWeather method returns. It returns a default sun WeatherCondition every time it is invoked. So at this point, reliably testing the logic of my controller is actually pretty difficult.

Coding to Interfaces

So what can we do to address these problems? A good option is to clean this code up, invert the control of instantiation of the WeatherForecaster instance, and take advantage of dependency injection. The first step is to refactor my code a little to make it easier to progress towards my final goal. The dependency inversion principle recommends that our classes rely on abstractions rather than implementations. Currently my HomeController is tightly coupled to the WeatherForecaster implementation. I'll now show you how I can refactor this application by creating an abstraction to let me reduce the coupling in my code. I'll do this by extracting a simple interface. I'll use the built-in Visual Studio refactoring tools to do this. Here I've right-clicked on the class name for the WeatherForecaster and chosen the Quick Actions and Refactorings menu item. And I'll choose Extract Interface from the Refactoring menu. In the dialog, I'll accept the defaults by clicking OK. I now have a new IWeatherForecaster interface defined in the application. The WeatherForecaster class has been updated by the tooling to declare that it implements this new interface. This is my first improvement as it means that the HomeController can now depend on this interface rather than a concrete implementation. This helps keep my code more loosely coupled. If I want to, I can declare multiple implementations against the abstraction I've just created. I can update my AmazingWeatherForecaster to implement the new interface since it has the required GetCurrentWeather method.

Inverting Control with Constructor Injection

With the preparatory refactoring complete, it's time to move on to using constructor injection. Constructor injection provides a way for dependencies to be passed into the class via the constructor rather than being created inside the class itself. This is achieved by making all dependencies of the class parameters of a constructor. I'll now show you how I can refactor my controller to implement constructor injection. I'm going to paste in some new code to add a constructor for the HomeController. The constructor accepts an IWeatherForecaster parameter, and it stores the dependency into a private readonly field. The use of the readonly key word is a good practice as it avoids the possibility of other methods from accidentally assigning a different value for the dependency after the controller is instantiated. My code now supports the dependency injection pattern since we allow the passing of any dependencies to this consuming code when it's constructed. Next, I need to update the Index method to use the IWeatherForecaster that was provided via the constructor rather than creating its own instance. At this point, I'm going to run the application, but I'm expecting it to fail as you'll soon see. As expected, an invalid operation exception has been thrown within the framework. The error message gives us a hint as to the cause of the problem. While attempting to activate the HomeController, a service for IWeatherForecaster could not be resolved. The reason I've got this exception is that I haven't configured the application so that the framework knows how to resolve the dependency. This is actually a quite common oversight, so don't feel bad if you see this occasionally when developing your applications. This is a runtime exception and not something that causes a warning or an error at compile time. To fix this exception, I'll need to make use of the built-in Microsoft dependency injection container.

Registering Your First Service

With the code refactored to support constructor injection, we are ready to register our first service with the Microsoft dependency injection container. The dependency injection container must be configured when the application starts by registering services with it. In ASP.NET Core, we use an IServiceCollection to register our application services. Any services which we expect to be injected by the container must be added to the IServiceCollection. The services you register with the IServiceCollection would like to be resolvable from the IServiceProvider once it has been built. In ASP.NET Core, the most common place to register services with the container is in the ConfigureServices method of the Startup class. I will now demonstrate how to register your first service with the Microsoft dependency injection container. We will locate the ConfigureServices method and then register our IWeatherForecaster with the IServiceCollection. The Startup class is included in all ASP.NET Core projects by default and is located in the root of your project. By default, Startup has two methods, Configure, which is used to set up the application middleware pipeline, and ConfigureServices, which I will use to configure my dependency injection container. Configure Services accepts an IServiceCollection as a parameter. This will be initialized and made available by the webhost when the application starts up. To configure the built-in container with my application dependencies, I will register them with the IServiceCollection. Various extension methods exist on the IServiceCollection to support different registration scenarios. Currently, the AddMvc extension method provided by the MVC framework is used to register all of the services needed to support using ASP.NET Core MVC in this application. The order of the service registrations is generally not important. They can occur anywhere within the ConfigureServices method. One situation where order will be a concern is if you find yourself registering multiple implementations for an interface. We'll discuss this in more detail in the module called Registering More Complex Services. My preference is to register my own application services at the top of the method before the framework and third party services on register service for the IWeatherForecaster using an extension method on the I Service Collection called AddTransient. This registers the service with a transient lifetime. I'll discuss service lifetimes in more detail in the module called The Microsoft Dependency Injection Container. The AddTransient overload I'm using here accepts two generic arguments. The first, in this case, is the abstract type, the IWeatherForecaster interface. This is a type, which application code can depend on as we saw in the constructor for the HomeController. The second parameter is the implementation type, which, in this case, is the concrete WeatherForecaster class. This is the type of the object that the container will create and return when resolving a request for an IWeatherForecaster. This simple line is all that is needed to register the service, so I'll run the application again. This time, no exception is thrown, and the home page loads correctly. The framework has been able to activate the HomeController by resolving the required dependency from the dependency injection container. With these changes, I've implemented the inversion of control principle. Our HomeController is now no longer responsible for creating an instance of the WeatherForecaster dependency. That responsibility now lies with the framework and is handled by the dependency injection container. Since my controller now depends upon an abstraction defined by the IWeatherForecaster interface, it is now decoupled from changes to the implementation, which gets injected.

Benefits of Dependency Injection

Dependency injection brings with it many benefits, which help make applications easier to maintain and evolve. Benefits include promoting the loose coupling of components and the use of abstractions, both of which help ensure that low-level changes don't propagate up to high-level code. Using dependency injection also supports improved testing while allowing unit tests to inject mocked dependencies. And it helps improve the readability and cleanliness of our code. Remember that amazing WeatherForecaster I created earlier, which also implements the IWeatherForecaster interface? Now that I'm using dependency injection, it's easy for me to swap out the implementation type, which will be resolved at runtime from the container by updating the service registration. Now when IWeatherForecaster is resolved and injected into the HomeController, it receives the new amazing WeatherForecaster without me modifying the HomeController class. A single change to my registration conveniently takes effect throughout my whole application for any code dependent on either the forecaster from the container. A second benefit is that the HomeController is now easier to test. Here I've updated by tests that I created earlier. Since the HomeController's constructor accepts an abstraction in the form of IWeatherForecaster, I could use a mocking framework, such as Moq, to supply a mock implementation. This allows me to set up the mock so that the GetCurrentWeather method returns a specific result every time that the test runs. Now I can easily assert that the Index method I'm testing returns the expected data on the HomeViewModel for each weather result that may be returned by the GetCurrentWeather method on the IWeatherForecaster. I'm now able to successfully run my unit test and properly test the logic of my HomeController.

Review

In this module, we've covered some important concepts that will enable you to start taking advantage of dependency injection in your ASP.NET Core applications. We started by refactoring an ASP.NET Core application to support loose coupling and apply clean architect patterns and principles. We looked at how we can inject dependencies into our code via constructor injection and how to register services with the built-in dependency injection container using the IServiceCollection. We concluded by reviewing the benefits these changes introduced, such as the ability to swap out implementations without modifying high- level consuming code and making our code more testable by allowing mock dependencies to be used in unit tests. In the next module, we'll look more closely at the Microsoft dependency injection container and identify some things to consider when registering services with it.

The Microsoft Dependency Injection Container

Introduction and Overview

In this module, I'll introduce the Microsoft dependency injection container, which is the backbone of all ASP.NET Core applications. You'll learn how this is used by ASP.NET Core to activate framework components, and I'll discuss the types of objects, which should be registered with the container, as well as those which shouldn't. Finally, you'll learn about service lifetimes, how these affect the services managed by the container, and how you should choose between them.

The Microsoft Dependency Injection Container

ASP.NET Core ships with its own built-in dependency injection container, which it uses to resolve the services it needs during the request process. All of the framework services, for example logging, configuration, and routing, use dependency injection, and they are registered with a dependency injection container when the application webhost is built. Internally, the ASP.NET Core framework is responsible for providing the dependencies required when activating framework components, such as controllers. An HTTP request is received via the ASP.NET Core web server, Kestrel. This is passed to the MVC middleware, which locates a suitable controller that can handle the request. ASP.NET Core activates its framework components as part of the request lifecycle using a helper class called ActivatorUtilities. These framework components, for example controllers, are not registered in the dependency injection container directly. During activation, any services required to create a controller instance are expected to be resolvable from the dependency injection container. The Microsoft container is packaged into its own assembly in the Microsoft.Extensions .DependencyInjection namespace. In the default ASP.NET Core templates, applications reference the Microsoft.AspNetCore .app meta package, which includes a reference to the dependency injection library. A dependency injection container, sometimes referred to as an inversion of control or IOC container, is a software component which manages the instantiation and configuration of objects. A dependency injection container is not a requirement in order to apply the dependency injection pattern. But as your application grows, using one can simplify the management of all of your dependencies, including their lifetimes. Services are registered with the container at startup and resolve from the container at runtime whenever they are required. The container is responsible for creating and disposing of instances of the required services on demand, maintaining them for a specified lifetime. The Microsoft dependency injection container is intentionally limited to a set of features required by the ASP.NET Core framework. However, as you'll see in the Registering More Complex Services module, there are some hidden powers for more complex scenarios. There are two main interfaces that we code against when using the Microsoft dependency injection container. The IServiceCollection interface defines a contract for registering and configuring a collection of service descriptors. It will be used to build an IServiceProvider, which defines a mechanism for retrieving a required service at runtime.

What to Register with the D.I. Container

Now that we've seen how to register services with the dependency injection container, let's discuss what types are suitable candidates to be registered. A good starting point is to review your code and look for places where the new keyword is used within framework components, such as MVC controllers. Once you find the new keyword, you must determine whether the object that is being created is actually a dependency. If the controller calls methods on the object, and the object is required in order for the consuming code to perform its responsibility, then your controller is dependent upon it. Once you identify a dependency, you can apply the dependency inversion principle and require that the dependency be injected via the constructor. Remember that you must also register the service with the container and that failing to do so will result in an exception being thrown at runtime. Once you have identified and registered a service, it's important to consider any dependencies that it has of its own, repeating the process when necessary. This hierarchy of dependent services is referred to as a dependency graph. If you adhere closely to the single responsibility principle, it's likely that you'll find you'll have many layers of dependencies in the graph as your application grows. As long as you make sure to register all of the services in the graph with the container and use constructor injection to accept the resolved dependencies, the container will do the work managing and creating the object graph of required services. In some cases, you will find usages of the new keyword that don't identify a dependency. This code in the HomeController creates a new HomeViewModel object. It is created by the Index method in order to return an output and is used to transfer data between layers of the application, in this case between the controller and the view. The HomeViewModel instance is passed to the View so that the data in the model can be used when rendering the page. The ViewModel class itself has no dependencies nor any logic. We often refer to classes that look like this as POCOs, plain old CLR objects. In this case, whilst it would technically be possible to register this class with the dependency injection container, that would be a misuse of the dependency injection pattern. This is not a dependency upon which the Index method relies, but really an output of the method. So it's perfectly fine to create it within the Index method. Another way to think about this is whether the object being affects the testability of the Index method. Do the properties or the methods of the object affect the logical behavior of a class under test? If not, then it's likely not a dependency that you need to inject. It may also be tempting to attempt to store primitive types, such as an integer or strings in a dependency injection container. These are poor candidates for dependency injection. Typically, they're not real dependencies of a class, although it's possible that some classes will accept them in their constructor. A common reason for requiring permitted types as parameters in the constructor of a class is to provide configuration. Value types such as integers and bools are not supported for registration at all using the generic registration methods on the IServiceCollection. Since strings are reference types, it is technically possible to register a string with a container using the generic methods, but this should be avoided. For these situations, it is far better to use the ASP.NET Core configuration system with a strongly typed options pattern.

Accessing Configuration from a Controller

Let's take a look at a best practice for dealing with configuration data in ASP.NET Core where we'll inject configuration options into the controller. To do this, we will use the built-in configuration system in ASP.NET Core along with the supported options pattern. For a more complete explanation of these features, I recommend that you check out the course ASP.NET Core Fundamentals. In ASP.NET Core, the options pattern is implemented with the help of the dependency injection container. With the options pattern, you define a simple POCO class to hold a set of related settings. Here's my class which I'll use to store a set of simple feature flags for my application. Each property will match the key of a configuration setting available in my application. This is my appsettings.json file, which holds some configuration values for my application. I have a Features section and within that a setting with the key EnableWeatherForecast, which matches the property in my Features configuration class. Next I must bind the configuration to my POCO class in order to use it with the options pattern. In ConfigureServices, you combined a section of your configuration to your options class using the ConfigureExtension method on the IServiceCollection. This strongly typed options class is now available via dependency injection. Once bound, I can inject an instance of IOptions of T into any classes which need access to configuration values. In my controller, I've added an IOptions of FeaturesConfiguration constructor parameter. I store its value, which will be a FeaturesConfiguration instance into a field. Now my code can make use of this configuration value as part of its logic. Since this is an injected dependency, I can easily mock this when unit testing the class to test my logic under different conditions.

Service Lifetimes

When registering services with a container, a service lifetime must be chosen for the service. The service lifetime controls how long a result object will live for after it has been created by the container. The lifetime can be defined by using the appropriate extension method on the IServiceCollection when registering the service. There are three lifetimes available for use with the Microsoft dependency injection container; transient, singleton, and scoped. The dependency injection container keeps track of all instances of the services it creates, and they are disposed of or released for garage collection once their lifetime has ended. The chosen lifetime affects whether the same instance of a service may be resolved and injected into more than one consumer. It is important to choose the lifetime of services wisely. We'll discuss each of the lifetimes and identify some situations when it is appropriate to choose them. As with many things in software development, there are no absolutes, and many conditions can affect the best choices, but I'll provide some general guidance. As we explore each lifetime, I'll use this simple API application to demonstrate the consequences of each choice. I have a GUID service, which creates and stores a new GUID when it is constructed. It returns the stored GUID as a string when the GetGuid method is called. I have some custom middleware, which uses the GUID service when handling each request. It gets the GUID from the GuidService and logs it to the console. I also inject the GuidService into the HomeController and log out the GUID value when the index action is invoked. By inspecting the GUID sent to the console, we'll be able to see how many instances of the service are resolved when using the different lifetimes. If you want to try this out for yourself, I've included the sample application in the exercise files for this course.

Transient Services

When a service is registered as transient, a new instance of that service is created and returned by the container every time that the service is resolved. In other words, every dependent class that accepts a transient service for our injection from the container will receive its own unique instance. This I most useful when the service contains a mutable state and is not considered thread-safe. Since each dependent class receives its own new instance, methods can be called on the service which affect its state without fear of access by other consumers. However, this can come with a small performance cost because it's likely that the object will need to be created multiple times during the life of the application. Every time it is resolved, which could be as often as every request, memory will need to be allocated. This in turn will lead to additional work for the garbage collector to clean up all of those short-lived objects. Transient services are the easiest to reason about because instances are not shared. Therefore, they tend to be the safest choice when it's not clear what lifetime is the best option when registering a service. I've set up my application to use the transient service registration for the GuidService. Here's the console output of the one request. The GUIDs are different because a new instance of the GuidService has been returned by the container each time it is asked to resolve it, once for the middleware and once for the controller. If I make a second request before the two new GUIDs are returned, I have two new instances of the GuidService. So even in this short demonstration, four instances have had to be created.

Singleton Services

An application service registered with a singleton lifetime will be created only once during the lifetime of the application. Strictly speaking, it's once for the lifetime of the root container. But in ASP.NET Core, these are one and the same. The same instance will be reused and injected into all dependent classes. If the service is required frequently, such as per request, then this can have an improvement on application performance by avoiding regularly allocating new objects, which then require garbage collection only moments later. Additionally, if the object is expensive to construct, limiting this to a single instantiation can improve your application performance versus registering the service as transient. When choosing to register a service with a singleton lifetime, you must consider thread safety. Since the same instance of a singleton service can be used by multiple requests concurrently, accessing or mutating state by consumers is likely to lead to unexpected behavior. Singletons are very well suited to functional style services where the methods take an input and return an output, and no shared state is used. A reasonable case for using the singleton lifetime is a memory cache, for example, where the state must be shared for the cache to function. But appropriate thread- safe programming techniques must be used in that situation. When registering services as singletons, consider the implications of a single instance remaining allocated for the lifetime of the application. It's possible to create memory lakes if large amounts of memory are held by a service given that this will never be released for garbage collection. If a service with high memory requirements is used very infrequently, the singleton lifetime may not be the most appropriate choice of lifetime. I've updated my sample to use the singleton lifetime for the GuidService. After one request, the same GUID is logged from both the middleware and the controller. If I make a second request, the same GUID continues to be logged. A single instance of the GuidService is being resolved in each case, even between separate requests.

Scoped Services

Scoped services sit in a middle ground between transient and singleton. An instance of a scoped service lives for the length of the scope from which it is resolved. In ASP.NET Core, a scope is created within your application for each request that it handles. Any scoped services will be created once per scope to act in a similar way to singleton services within the context of that scope. All framework components, such as middleware and MVC controllers, will get this same instance of a scoped service when handling a particular request. If you use Entity Framework Core in your application, by default the DbContext is registered with a scoped lifetime. Any services which depend on a scoped service should themselves be registered as either scoped or transient to avoid the object being captured beyond its expected lifetime. I've updated my sample once again, this time using the scope lifetime for the GuidService. After the first request, the same GUID is logged from both the middleware and the controller. If I make a second request, it has a new scope. So this time whilst the GUID is the same between the middleware and the controller, it's different from the GUID in the first request.

Avoiding Captive Dependencies

An important point to consider when registering your dependencies is to ensure that the chosen lifetime is appropriate, taking into account any dependencies that the service has of its own. This is necessary to avoid captive dependencies, which is where a service may live for longer than is intended. A service should not depend on a service with a lifetime shorter than its own. For example, a service registered with a singleton lifetime should not depend upon a transient service. Doing so would result in the transient service being captured by the singleton service with the instance unintentionally being referenced for the life of the application. This can lead to problematic and sometimes hard-to-track-down runtime bugs and behaviors, such as accidentally sharing non-thread-safe services between threads or allowing objects to live past their intended lifetime. To visualize this, let's consider which lifetimes can safely depend on services registered using another lifetime. Since it is short-lived, a transient service can safely depend on services that have the transient, scoped, or singleton lifetimes. A scoped service should not depend on a transient service as it may be unintentionally allowed to be reused by multiple consumers within a single request. It is safe for a scoped service to depend on other scoped or singleton services. A singleton service is the most restrictive in terms of its dependencies, and it should not depend on transient or scoped services, but it can depend on other singleton services. In ASP.NET Core 2.0, a feature was added called Scope Validation. This is enabled by default when the environment is set to development. It validates the container configuration to ensure that no scoped services are captured within singleton services. When enabled, validation will occur at application startup. If any captive scoped services are identified, a runtime invalid operation exception will be thrown. This feature is not enabled in the production environment by default since it may have performance implications. If you wish to change this behavior, you can configure the ServiceProvider options when configuring the webhost builder in a program class. But remember, in the production environment, it is not recommended to run in this configuration.

Review

In this module, we talked about the built- in Microsoft dependency injection container. You've learned about the types which are suitable for registration with the container. We've seen how we can inject configuration into our code using the options pattern, and you're now familiar with the different service lifetimes and some guidelines for choosing between them when registering your services. Join me in the next module where we'll build upon this knowledge, dive deeper into service registration techniques, and tackle more complex registration scenarios.

Registering More Complex Services

Introduction and Overview

Welcome back. In this module, we're going to take a deeper look at registering services. You'll learn about some of the extension methods on the IServiceCollection and how to choose between them. I'll cover some situations where it may not be readily apparent how you can register services for some scenarios you may face when writing your applications. For each situation, I'll review the options available when using the Microsoft container and start to explore where we'll run into its feature limitations. You'll also learn how to improve the service registration code organization in larger and more complicated applications.

Introducing the Tennis Booking Application

Until this point, I've used a fairly simple tennis club website to demonstrate registering services with a dependency injection container. For the remainder of this course, we'll develop using this more advanced version of the website. It has some more complex requirements, which support demonstrations of tackling more complex service registrations like those you may encounter in your own applications. You'll see various different stages in the application's development as we progress through the remaining modules. The key points that I want you to take away from this sample are the code examples of registering and resolving services from the container. You can download the completed application from the course downloads. This final version of the site includes functionality for its members to book courts at the club. The member-facing website now uses as ASP.NET Core Razor pages programming model. Club administrators can manage the court bookings for the club using the administrator site. The administration area of the website is built using a traditional MVC approach.

Service Descriptors

Before we dive into the various registration scenarios, it's worth a short interlude at this point to touch on the concept of service descriptors. Service descriptors contain the information about services, which have been registered with the dependency injection container, and they're used internally by both the IServiceCollection and the IServiceProvider. The ServiceDescriptor class includes properties, which hold details about registered services. This includes the ServiceType for the registered service, which is typically going to be an interface, details about the implementation type, which will be the type constructed when the service is resolved, and information about the lifetime of the service. It's rare that you should need to work directly with service descriptors since they are typically created for you by the various extension methods on the IServiceCollection. However, as I'll demonstrate later in this module, there are some situations where you will need to work more directly with service descriptors. Once built, the service provider and its related internal components use a collection of service descriptors for your application based on any registrations made against the IServiceCollection. Let's take a quick look at a few options for working with service descriptors using the tennis bookings application as an example. Currently, I'm using the generic AddSingleton extension method to register the IWeatherForecaster service, and this is very convenient. I'll comment out that code for a moment and paste in four examples, which each create a service descriptor for this service. Each of these results in an equivalent service descriptor. The first option creates a new instance of a ServiceDescriptor directly and passes in three arguments. The first is the service type, the IWeatherForecaster interface in this case. This is the type that other services can depend on. The second is the implementation type, the WeatherForecaster class in this case. And finally, I'll choose a lifetime from the ServiceLifetime enum. The second option here uses a static Describe method, which exists on the ServiceDescriptor class. This then accepts the same three arguments. Since we always need to choose a lifetime when we describe a service, static methods exist for each lifetime, which simplify the creation of a service descriptor. Here I can use the Singleton static method, which means that I need to pass the service type and the implementation type. A generic version also exists, which is cleaner since we no longer need to use the typeof expression. Once you have a ServiceDescriptor instance, you can add it to the service collection directly. I want to reiterate though that in most situations, you should use the extension methods on the IServiceCollection and avoid creating service descriptors directly.

Add vs. TryAdd

We're seeing how we can register services with a dependency injection container. What happens if you create more than one registration for the same service type. In the next demo, we'll put this to the test and find out. We'll then explore the difference in behavior between the Add and TryAdd extension methods on the IServiceCollection. The Microsoft dependency injection container supports a configuration as code approach. When registering services with the IServiceCollection, there are many extension methods that you can choose from. The most common are the generic AddTransient, AddScoped, and AddSingleton methods, which we've seen earlier in the course. Each of these have various overloads, accepting different arguments and allow you to add your service to the IServiceCollection with a specified lifetime. Here the I WeatherForecaster has been registered with an implementation from the WeatherForecaster class. I'll create a second registration for the IWeatherForecaster interface known also as the service type, this time using the Amazing WeatherForecaster as the implementation. I'll add a breakpoint in the constructor of the page model for the index page. If I run the application, we can inspect what implementation type gets injected by the dependency injection container. The implementation type is the AmazingWeatherForecaster. In situations where there are multiple registrations for the same service type, it is the last registration which wins. This is, therefore, a situation where the order of the codes used to register services in the ConfigureServices method can be quite important. As your applications get more complex and you have a lot of service registrations, accidents can happen, and it's possible for duplicate registrations to creep into your code. We can avoid errors by making use of some other extension methods on the IServiceCollection. I will now change the registration using the AmazingWeatherForecaster implementation to use the TryAddSingleton method. Only the method needs to be changed here. The arguments remain the same. To call the TryAdd extension methods, a using statement is required for Microsoft.Extensions .Dependency Injection.Extensions. After this small change, if I run the application, we hit the breakpoint again. This time the injected type is the WeatherForecaster, the first service I registered. When using TryAdd, the method will only register a service when there is no implementation already defined for that service type. The second call to TryAdd attempted to register a second implementation for IWeatherForecaster, But since that registration already existed, this registration was skipped. This is convenient when you have more complex applications because you can Express your intent more clearly when registering your services and avoid accidentally replacing previous registered services. Unless you explicitly know you wish to register something, even if a prior registration exists, it's often safest to consider using the TryAdd methods.

Registering an Interface Multiple Times

When I called the AddSingleton method multiple times for the IWeatherForecaster, we saw that the dependency that the HomeController received was the AmazingWeatherForecaster, the second and last implementation that I registered. You may now be wondering what happened to the first registration in that scenario. A reasonable assumption would be that perhaps the second registration entirely replaced the first one, but that's not actually the case. Let's take another look at the code and see what is actually happening. I'll revert back to the code without the TryAddSingleton registration so that I have two registrations for the IWeatherForecaster service type, both using the AddSingleton method. I'll add a breakpoint here on the second registration. After running the application, when the breakpoint is hit, we can inspect the state of the IServiceCollection. The IServiceCollection implements the IList of service descriptor generic interface, so we can explore the current items in the Locals window. Here we can see an entry in index 68 for the IWeatherForecaster with the WeatherForecaster as its implementation type. If I step over this line, allowing the second registration to occur, we can inspect the service collection again. This time, the last entry is at index 69, a registration for IWeatherForecaster, this time with AmazingWeatherForecaster as its implementation type. Just above this, we can see that the first registration at index 68 is still available. Coding an Add method, such as AddSingleton, multiple times for the same interface results in multiple entries in the service collection. When a service depending on IWeatherForecaster is created, the dependency must be resolved from the container. The last registration into the container is assumed to be the preferred choice and therefore an instance of its implementation type will be resolved.

Replacing and Removing Registrations

Having looked at the behavior of the Add and TryAdd methods, there is one more scenario which remains. What if we do actually want to replace prior registrations? In this demo, you'll learn how to replace a service registration. I'll also show you how we can remove service registrations from the IServiceCollection. I'll now update the registrations using AmazingWeatherForecaster to replace the previously registered implementation type for the IWeatherForecaster service type. The IServiceCollection supports calling a Replace method, which will do exactly that. The Replace method does not support any generic overloads, and so this is one of the rare situations where we must provide a ServiceDescriptor argument. I'll use the Singleton static method on the ServiceDescriptor, which lets me define the descriptor for the IWeatherForecaster service type with the AmazingWeather service implementation. The Replace method will look for the first service registration for the IWeatherForecaster service type, and if it finds one, remove it. The new implementation type will then be used to create a new registration in the service collection for the IWeatherForecaster service type. If I add a breakpoint and run the code, we can inspect the items in the service collection. This time only the AmazingWeatherForecaster implementation exists because it has fully replaced the previous registration. Remember though, the replace option only supports removing the first registration of the service type in the collection. In cases where you want to remove all prior registrations of the service type, you can call the RemoveAll method, passing the type for the service that you wish to remove. If I undo that last change back to the point where I have two implementations registered, we can see the effect of the RemoveAll method. The RemoveAll method supports a single generic argument, which is the service type that we want to remove. For this example, I'll remove all IWeatherForecaster implementations. Now if we hit the breakpoint and inspect the IServiceCollection, we can see that both registrations against IWeatherForecaster have been completely removed. The last registration in the IServiceCollection at this point is now the IWeatherApiClient, which is the registration line above both of my IWeatherForecaster registrations. I expect both of these scenarios to be quite rare, but they may prove useful if you want to provide your own implementation for framework with third party services, for example.

Registering Multiple Implementations of an Interface

As we've seen, when you register multiple implementations for service using the Add methods, each registration will be stored in the service collection. So why does the container store all registered implementations when it will only resolve the last one? In this demo, we'll look at a use case for registering multiple implementations of an interface or service type. But we'll use this to our advantage to support adding functionality to an application with minimal changes to existing code. There are situations where it might actually be useful to have multiple implementations registered against the same interface. One such example is when applying a rule pattern. In the tennis booking application, members can make court bookings. Before the booking is completed, the booking is evaluated to ensure that it is valid given the club rules for bookings. When a member tries to make a booking which fails one or more of these rules, the booking attempt is rejected. To keep the application flexible in regards to the addition and removal of rules in the future, I've created an abstraction, which defines a court booking rule. It includes a method called CompliesWithRuleAsync, which takes a court booking and returns a bool, indicating if the court booking complied with the rule or not. In my ConfigureServices method, I have four rules defined, which implement this interface. The registrations are using the AddSingleton and AddScoped registration methods, and therefore each registration will be added to the service collection, even if existing registrations for the service type exist. It's fine to use registrations with different lifetimes, which are appropriate for the implementation type being registered. So how can I now utilize the multiple implementations I've registered? The BookingRuleProcessor brings these rules together, allowing all of the rules to be evaluated. In its constructor, it accepts an IEnumerable of ICourtBookingRule. When the dependency injection container is resolving services for this class, it will now attempt to resolve all instances of ICourtBookingRule and inject them as an IEnumerable. Note that the container will only do this when a parameter is IEnumerable. It does not resolve correctly in situations where you specify an array or any of the other collection types. In those cases, specifying them as parameters would cause an invalid operation exception to be thrown. The RuleProcessor can now loop through all of the injected rules, evaluating the court booking against each of them. With this approach, it allows me to add additional rules very easily without needing to change the processor or any of the existing rules which have been defined. Each rule is independent and therefore has a single responsibility. At the moment, a member can make two bookings, which occur at the same time on the same date. This is clearly not valid, so I'll add a new rule to evaluate this. I'll create a new class named MemberBookingsMustNotOverlapRule, which implements the ICourtBookingRule interface. I then need to implement the required members. For the ErrorMessage property, I'll return a suitable error message. Next, I'll add a constructor, which accepts an ICourtBookingService dependency. I'll use this service to load the court bookings for the current club member, which will be needed to evaluate this rule. Finally, I'll paste in some code that will implement the required logic for the rule. It gets all of the bookings for a user for the date of the booking being made and checks whether any of the booked hours overlap those in the new booking, which is being added. With the rule added, the only other thing I need to do is to register this service in the ConfigureServices method. I'll add a scoped registration after the other rules to register this new one. I'm using scoped in this case since I know that the ICourtBookingService, which my new rule depends on, is also a scoped service. If I now run the application, it will test whether this new rule is taking effect. I already have a booking for this member on Court 1. If I attempt to put Court 2 at the same time, I now get the expected booking failure because it fails the new rule. I've added a new rule here without touching any existing business logic. The only update to existing code that I've had to make is to add the registration of the new rule to the IServiceCollection. A downside of this approach for registering multiple rules is that each implementation must be manually registered with the container. If I forget to register the rule, then it will never be applied by the RuleProcessor. More feature-rich dependency injection containers support a feature called assembly scanning, which can automatically locate and register all implementations of a given interface. In that scenario, just by adding the new implementation of the rule, it would be found and registered by the assembly scanning process automatically. This is an example of where the limited feature set of the Microsoft container may be a little frustrating. In the module called Beyond the Built-In Container, we'll explore some options to address this.

Improving Multiple Implementations with TryAddEnumerable

In the last clip, we looked at a use case for registering multiple implementations of an interface. There is an issue with the current approach that I want to highlight. What happens if the same implementation is added twice? First, we'll test what happens by duplicating one of the rules and reviewing the registrations with the IServiceCollection. Then we'll look at an alternative registration using the TryAddEnumerable method, which provides slightly different behavior. I'll start by adding a duplicate registration for the MembersBookingsMustNotOverlapRule. If I add a breakpoint in the BookingRuleProcessor, we'll be able to see what rules are injected by running the application. I'll make a new booking, confirm the booking length, which will trigger the rule processing before the booking can be saved. If we inspect the IEnumerable of ICourtBookingRule that has been resolved, we're now seeing six rules. The MembersBookingsMustNotOverlapRule is included twice. The reason for this is that the Add methods are not idempotent and therefore will allow the same implementation for an interface to be registered multiple times. In this case, it's not a major problem for this rule to be executed twice because the rules are stateless and have no side effects. However, it's not very efficient to have duplicate instances of the same rule, each being evaluated by the RuleProcessor. Our situation could be more dangerous as duplicated instances could easily have unplanned side effects. Therefore, when registering multiple implementations for an interface, it's recommended that we use the TryAddEnumerable method, which prevents duplicate registrations of the same implementation. I'll paste in an improved version of the registrations, which take advantage of the TryAddEnumerable approach. Calling the TryAddEnumerable method is another example of where we need to work directly with service descriptors. Here I'm using the appropriate static methods to create ServiceDescriptor instances. I've left in place a duplicate registration of the MemberBookingsMustNotOverlapRule, this time using the TryAddEnumerable method. Let's run the application and check the resolved rules. Now there are only five registered rules, and there is no duplication. Much like the other TryAdd methods, using TryAddEnumerable is recommended to avoid unintended duplication and to better Express your intent when registering services. If you prefer, there is an overload of the TryAddEnumerable method, which accepts an IEnumerable of ServiceDescriptor. This provides a slightly terser syntax for registering multiple implementations. Here I'm passing an array of ServiceDescriptors to register my five rules in a single method call.

Implementation Factories

In some advanced situations, the creation of the implementation type for a service may require some manual intervention. Implementation factories offer more control over how the implementation type is created. In this demo, we'll see how we can register a service using an implementation factory. We'll also discuss some of the scenarios where it may be necessary to use them. When adding a registration which provides an implementation factory, we provide a delegate, which will be called when resolving the service. This delegate is responsible for creating an instance of the service type. For example, where a builder of factory pattern is used to create an object, we may need to use an implementation factory. Our first choice in these situations should be to review where your code can be refactored to better support automatic creation by the container. But this may not always be possible when working with legacy or third party code. The first example we'll look at for implementation factories is forwarding a registration of the service type onto an existing registration. A common preference I've seen with developers working on ASP.NET Core applications is to allow access to configuration values without taking a direct dependency on IOptions of T. We look to injecting configuration using IOptions of T in the module called The Microsoft Dependency Injection Container. Here I have a BookingConfiguration class defined. Currently, this is bound via the options pattern from the application configuration. I'd like to support injecting this directly via an interface to any dependent classes. First, I'll extract an interface for this booking configuration. I'll now add a singleton registration into the dependency injection container. If I were to create this registration using the BookingConfiguration class as the implementation type, when resolved for the first time, a new instance of BookingConfiguration would be constructed, and its properties would have default values. But this is not what I want since I want to be able to access the previously bound configuration values. I can take advantage of forwarding to return an existing object from the dependency injection container for this service type. I've created a registration using a single generic argument, which provides the service type of the IBookingConfiguration interface. I've used an overload, which accepts an implementation factory as an argument. This is a func with a single parameter, in this case the final IServiceProvider. The delegate will be invoked once the service provider has been built, and therefore we have access to it in the implementation factory, allowing access to previously registered services. The func returns an implementation instance of the service type for the registration. I have provided a lambda expression, which uses the service provider to resolve the IOptions of BookingConfiguration from the service provider. I can return the value of this, which will be the BookingConfiguration instance, which has the configuration values bound to it. I have now created an additional registration using a different service type that allows access to the same underlying object. I can now update the MaxBookingLengthRule to use this new interface as a constructor parameter. Another potential use case for implementation factories is to implement the composite pattern. Whilst the limited feature set of the Microsoft container doesn't support this pattern directly, we can register a non- generic composite by making use of the implementation factory approach. In the tennis booking application, once a booking is completed, a confirmation notification is sent to the user. The initial requirement is for the application to send a notification via email. But recently a requirement was defined, which requires that a notification also be sent via SMS. Currently the application defines an INotificationService interface, which the EmailNotificationService implements. This interface defines an SendAsync method, accepting a message and the IDE of the user to be notified. Dependent classes, such as the CourtBookingManager require an INotificationService to be injected, which they can then use to send a notification. Thanks to dependency injection, it's possible to fulfill the new requirement of sending an SMS message without having to change the dependent class. I've already added an SMS notification service to this application, which also implements the INotificationService interface. Additionally, I've added a CompositeNotificationService, also implementing the interface. This service accepts an IEnumerable of INotificationService as a parameter of its constructor. Its SendAsync method calls the SendAsync method on all of the instances which it receives. I'll now update the service registration to support a basic composite pattern here. The SMS and email service are now registered against their own types and not against the INotificationService interface. I've added a new registration for the INotificationService interface, which passes a lambda as the implementation factory argument. The delegate constructs a new CompositeNotificationService. To this it passes an array of INotificationService items, which it populates using the service provider. In future, whenever an instance of INotificationServices resolve from the container, it is the CompositeNotificationService instance which will be returned. As we saw, this sends an email and SMS message via the respective services, and therefore the new requirement is now satisfied. Notice that I did not need to change any existing code except for my service registrations, which can make this a powerful way to extend applications. The final example, which I'll show demonstrates how we can use an implementation factory to return implementations for a service that cannot automatically be constructed by the service provider. Here I have an example where the IMembershipAdvert service is registered with an implementation factory that makes use of a builder class. First, I request the builder from the service provider, which is available at the time the service is being resolved. I then call a method to apply a discount of 10 pounds for this advert instance. And finally, I use the builder to construct the final membership advert. This approach is useful when working with legacy code or third party code. In these situations, you may want to add types to the container where you have no direct control over their creation, and perhaps you have to use a factory or builder to create instances of them. I consider the use of implementation factories a more advanced case that you should rarely need to rely upon.

Registering an Implementation against Multiple Service Types

In this demo, we'll look at where it can sometimes be useful to be able to register a single implementation against multiple service types. We'll take a look at a few potential techniques for achieving such registrations. You'll learn about the problems with some of the approaches and ultimately which choice results in the correct and most desired outcome. You may find yourself in situations where one of your services implements more than one interface and where you would like the service to be resolvable via each interface. In the tennis booking application, there is a service which provides a random greeting message to visitors on the home page. The greeting is provided by a GreetingService, which implements an IHomePageGreetingService interface. This service loads a collection of greetings from a file when the service is constructed and stores them in an array in memory. An additional random greeting feature is now needed, which greets logged in members with a personalized message when they access their bookings. The plan is to extend the greeting service to support both types of greeting. I have completed some of the steps already so that we can pick up at the point of updating the service registrations. Rather than change the existing interface, a new more generalized IGreetingService interface has been defined. I've already update the GreetingService to implement this interface. And you can assume it should accept an IGreetingService, and over time, I will factor out the old interface, which is now considered depreciated. The challenge here is registering the GreetingService implementation against both of these interfaces. A tempting option is to add an additional registration for the IGreetingService with the implementation type of GreetingService. I'll demonstrate why this is problematic by running the application. The home page loads, and we can see a greeting. For demonstration purposes, I've also included a GUID after the greeting. This is set once when the GreetingService is first constructed. Refreshing the home page a few times results in a new greeting, but the GUID remains the same. This is because the service is registered as a singleton, and therefore the GUID is generated once when the service is created, which happens the first time that this is resolved from the container. There the GUID starts with ae7a5. If I navigate to the bookings page, we'll see the new personalized greeting, which is defined in the updated interface. This is being injected via the new service registration that I've just added. Note though that the GUID starts with 0456b, which is different to the one on the home page. Since our service is a singleton, we might expect to get the same instance of the service and therefore the same GUID on both of the pages. Let's look at the registration code again to understand the problem. Because I've created two registrations, one for each interface, when each interface is resolved, its own GreetingService is created. Each of these instances will have a singleton lifetime, but they are unique per registration. In some situations, this may be acceptable. but here our GreetingService loads some greetings data into memory when it's constructed. Having two instances means that I have double the memory usage because each instance maintains its own array of greetings. I want to ensure that the same singleton instance is resolved for both interfaces, and there are two ways I can tackle this. The first option is to use an overload of the TryAddSingleton method, which accepts passing an existing instance of an object as an argument. In the ConfigureServices method, I can create an instance of the GreetingService and pass it into both service registrations. The GreetingService constructor has a single parameter requiring an IHostingEnvironment to be provided. Fortunately, this is a fundamental service in ASP.NET Core, which is available by the time that this startup method is called. I've already injected this into the Startup class's constructor so that in this case I can make use of it to create the GreetingService. Now I have an instance of the GreetingService, I can pass this into both registrations so that they both return the same instance when they're resolved from the container. If I now run the application, we can check the result of making this change. On the home page, this time the GUID begins with f054e. And on the bookings page, it's the same. There is a now a single instance of the GreetingService when the application runs regardless of which interface is being used to resolve it from the container. A downside of this approach is that I'm now responsible for creating than instance of the GreetingService. In this example, my service has a single parameter needed to construct it. Fortunately, it's a type that's already available when the ConfigureService method is called. More often than not, this won't be the case for your services. Those services are likely to be other application services not available when this Startup method runs. So what can we do to improve the service registrations here? The answer is to make use of implementation factories, which we've looked at previously. First I'm going to create a registration where the service type and the implementation type are both GreetingService. This registers the GreetingService so that I can access it from the service provider directly. Next, I'm going to create two further registrations, one for each interface. These will use the implementation factory approach to forward the interfaces to the GreetingService via the service provider. Because the GreetingService is being resolved by the container, any dependencies required in its constructor, such as the IHostingEnvironment, can now be injected at runtime. This supports any number of dependencies as long as they are registered with the service collection in the usual way. This approach also supports the use of transient and scoped lifetimes, which would not have been possible with the previous approach where passing an existing instance forces the use of the singleton lifetime. Again, the situations where you should need to do this with your registrations should hopefully be quite rare.

Registering Open Generics

In this demo, we'll look at how to register an open generic service with the Microsoft dependency injection container. An example of this in the framework is the ILogger of T interface, which is implemented by ILogger of T. Classes can then depend on an ILogger of T where the generic argument is their own type. Each generic type gets its own instance, which is resolved at runtime. This allows each resolveLogger instance to use the type information to set the logging category we'll use. Registering these manually would be quite tedious, which is why the Microsoft container supports the registration of open generics. In the tennis booking application, I followed a similar pattern to register a wrapper around the distributed cache, which supports a common cache key prefix based on the type that's being cached. Here's my generic interface for an IDistributedCache of T. My implementation of this interface uses the generic type parameter to build a cache key prefix to ensure that each key is more unique within the cache. I want to register this so that consumers can request a DistributedCache instance for any object which they need to cache. This will then have the appropriate cache key set. Here's an example of a possible registration which uses closed generic arguments for the IDistributedCache service type and the DistributedCache implementation type. This is an explicit registration and works if I only have a single type that I wish to cache against. What happens if I need to cache other types? Using the current approach, I would need to manually add a new registration for each type that may possibly be cached. Here's the second registration for another type which needs to be cached within the application. Whilst this works, it means that every time someone decides to use the IDistributedCache for a new generic type, they also need to remember to register it. This is not the most practical solution for this situation. A better way to approach this problem is to utilize open generic registrations with a container. This allows us to define a single registration, which can handle any type argument for the IDistributedCache. And runtime, when a service is resolved, the generic type argument will be provided, and the container can create the appropriate generic implementation type. I'll past in the open generic registration that I'll use for this application. Because I'm not able to close the generic argument at registration time, I cannot make use of the generic IServiceCollection TryAddSingleton method. Instead I have to use the non-generic version, which accepts as parameters the service type and the implementation type. I use the type of expression to pass an open generic IDistributedCache service type for the first argument. For the implementation type, I can do the same thing with the DistributedCache type. Now I no longer need to worry about all of the possible cache types in the application, and dependent classes can have them injected as needed with the generic type closed by the container at runtime. I don't expect you to find yourself in many situations that require open generic registrations, but it's useful to be aware that they are then when you need them.

Clean Code Using Extension Methods

In this demo, we'll look at a technique of creating our own extension methods, which will improve the cleanliness and readability of the ConfigureServices method. As your application grows in complexity, you will likely find your ConfigureServices method becomes filled with many tens or even hundreds of lines of code needed to register all of your acquired services. This can make it hard to locate registrations and understand the composition of the dependency graph for your application. When you reach this stage, it's useful to spend some time improving the structural organization of your registrations. My ConfigureServices method, even in this relatively small sample application, is pretty complicated and has gotten quite long. This can make navigating through the services quite confusing, and it's easy for things to get lost in the noise. There are some quite distinct themes though with sequences of registrations, which tend to relate to common sets of functionality. Take, for example, these first few registrations, which all relate to the configuration of my application. I'm going to add a new class to contain an extension method on the IServiceCollection. I'll call it ConfigurationServiceCollectionExtensions. By default, this picks up a namespace for our application in the location of the class. A convention with the dependency injection extension methods is to include them in the Microsoft.Extensions .DependencyInjection namespace. This is particularly true when you're building the libraries, which includes some registration logic. It's not a requirement to do this though, so it comes down to personal preference. I'll make the class static and create a static extension method. This method will return an IServiceCollection so that I can support a fluent chaining of calls to add services. I've called this method AddAppConfiguration. The use of add followed by a description of the feature this method registers services for is a common naming convention. You'll be familiar calls at AddMvc for example, which is an extension method that registers all of the required MVC-based services. For now, the only parameter for this method will use the this key word to identify the type that this method is extending. In this case, I'm extending the IServiceCollection. With the skeleton of my method defined, I'll jump back to the Startup class and cut the group of registrations which related to configuration. I can then paste these into the new extension method I've just created. There's quite a lot of using statements needed here, which I'll go ahead and add. This extension method is made a little more complex because the configure methods require an IConfiguration object. For this extension method, I can allow that to be passed in as a parameter. Finally, the method signature says that this method will return an IServiceCollection. So I'll return the collection being operated on. Now I can update the Startup class to call the new extension method. And that's it. Eleven lines of code have been reduced to a single line, and already this Startup class is looking less cluttered. It's now a case of repeating the process for the rest of the registrations. I won't make you watch me refactor everything, and instead let's just jump to the final Startup class. Here I've added many more extension methods for logical groupings of registrations. Because I made my methods return an IServiceCollection, I can shorten my code further using a more fluent registration style. This has made the Startup class much less intimating to understand and read. After my application registrations, I've left the framework registrations in place. I could easily refactor these behind an additional extension method, but personally I find it's useful to leave these in place so that I can quickly check how the framework features have been configured. This approach is also very useful as a library offer, whether that be for public NuGet libraries or when building libraries internally to support common functionality between applications. By including an IServiceCollection extension method in your library, you can provide a simple way for consumers to ensure that all acquired services are properly registered.

Review

In this module, we have covered a lot of ground with many code examples. You've learned about some of the more complex scenarios, which you may encounter, and I've shown you various techniques to support registering services in those situations. In large enterprise applications, I expect you will find yourself drawing on some or all of the techniques that I've talked about. We learned about service descriptors, which some of the less common registration overloads expect. We've talked about declaring our intent using the TryAdd variants of the extension methods on the IServiceCollection. We've looked at dealing with multiple registrations of both service types and implementation types. And we've covered registering open generic services. Finally, you've learned about an important approach to improving the readability of your service registrations by creating your own extension methods on the IServiceCollection. You leave this module with a complete understanding of the capabilities of the built-in dependency injection container, and you will now be in a position to know when you've hit the limits of those capabilities. Join me in the next module where we'll explore how and where services can be injected and resolved in ASP.NET Core applications.

Injecting and Resolving Dependencies

Introduction and Overview

Welcome back to Dependency Injection in ASP.NET Core. In this module, we'll continue our journey by looking at injecting and resolving dependencies. We'll begin by learning about the places where services can be injected via constructor injection and the rules that apply when injecting it into constructors. We'll take a look at injecting services directly into action methods and why this may be useful. We'll learn about the two approaches to injecting services into middleware components. And we'll explore how to inject services directly into Razor views. We'll conclude by covering some more advanced topics that are relevant to resolving services in ASP.NET Core applications.

Service Resolution Mechanisms

ASP.NET Core resolves services via two mechanisms. The most common in many situations will be via the IServiceProvider directly. In this case, services must first have been registered with the container in the ConfigureServices method in the Startup class. When a dependent class requires an instance of a service, an instance will be resolved from the container. The second mechanism of activation is via the static ActivatorUtilities class. This supports creation of objects that have not been registered into the container via their static methods. ActivatorUtilities will attempt to create an instance of an object using its constructor. Arguments for the constructor can either be supplied directly, or they can be resolved from an IServiceProvider. The framework uses ActivatorUtilities under the hood to activate some of its framework components. Controllers, tag helpers, and mobile binders are just a few examples, and these are not registered into the container by default. You should rarely ever need to use this class directly in ASP.NET Core applications.

Constructor Injection

So far in this course, we've seen that dependencies can easily be injected via constructors. Constructors will likely be the most common place where services are injected and resolved in your applications. You can inject dependencies into the constructors of controllers, Razor page models, ViewComponents, TagHelpers, filters, and middleware. Any of your own classes which are registered with the container can also accept dependencies via their constructor. Constructor injection is very easy and convenient to use. There are, however, a few rules that must be observed when using constructor injection with the Microsoft dependency injection container in ASP.NET Core. If constructors accept arguments which are not provided by the dependency injection container, then default values must be assigned for them. The defaults will take effect in cases where the object is created by the container. Constructors must be public in order to support dependency injection via both the IServiceProvider and via the ActivatorUtilities class. In cases where an object is being activated by the framework using ActivatorUtilities, there can only be one constructor overload where all of the arguments can be fulfilled via dependency injection. To demonstrate this last rule, I'll jump back into the tennis booking sample. Razor page models are an example of things which are activated by the framework using ActivatorUtilities. Here in my index PageModel, I'll add a second constructor by copying the original one. In this copy, I'll add an additional dependency on an ILogger of T. Let's run the application and see what happens. We get an InvalidOperationException. The message explains the problem. I'm now breaking the rule about having more than one applicable constructor for my PageModel. Both constructors could be satisfied by the services registered with the dependency injection container, and therefore the framework cannot decide which is the appropriate constructor to use. Looking at the stack trace, we can see this error is thrown within ActivatorUtilities and its TryFindMatchingConstructor method. In contrast, services which have been resolved from the dependency injection container directly can support multiple applicable constructors. In this simple example, I have a service called MyService, which will be resolved from the container. It has two constructors. The first has a single parameter, and the second has two parameters. In this example, both can be satisfied by dependencies from the container. So the most parameter-rich constructor will be used to create the instance. If this was a situation where another service had not been registered with a container, then when resolving this service, the first constructor would still be valid and used to create an instance of MyService via the service provider.

Action Injection

Besides constructor injection, ASP.NET Core controllers support a second point of injection called action injection. In this demo, we'll look at the reasons why we might choose to use action injection in our applications. We'll then look at an example of refactoring an existing controller to make use of action injection. Dependencies identified for action injection are only injected if the action specifying the dependency is invoked to handle a request. Choosing between constructor injection and action injection comes down to how widely used the dependency is within a controller. In the admin area for the site, I have a CourtsController, which receives two dependencies for our constructor injection. The ICourtBookingService is the first injected dependency. If I select the backing field and scroll through the methods in this controller, we can see that it's used by the WeeklyBookings method, the CancelBooking method, and the CancelBookingConfirmation method. The last method in this controller called UpcomingMaintainence does not use the ICourtBookingService. Now let's look at the second injected dependency, the ICourtMaintenanceService. Scrolling down, we can see that this is only used by the UpcomingMaintenance action method. Because I'm using constructor injection for both of these dependencies, the ICourtMaintenanceService will be injected into the controller every time that it is activated regardless of which action is actually being invoked. The ICourtMaintenanceService is registered as a scoped service with a dependency injection container. Every time when your request comes in, which is rooted to an action in this controller, the controller will be activated by the framework, and a new instance of the ICourtMaintenanceService will need to be instantiated. This results in additional allocations for this object and additional work for the garbage collector to clean up the instance when the request completes. Since this service is only needed for one action in this controller, this is slightly wasteful for any of the requests which are rooted to any of the other action methods. One way to address the problem in this example is to make use of action injection for the ICourtMaintenanceService. To do that, I'll add a parameter for the UpcomingMaintenance method. I'll mock this parameter with the FromServices attribute. This identifies to the framework that this parameter should be resolved from the dependency injection container. This parameter is of type ICourtMaintenanceService, and I'll name it appropriately. I can update the action method to use the injected dependency from the method parameter. Since I don't use this service elsewhere, I can remove it from the constructor parameters and get rid of the backing field to which it had been assigned. Now when a request roots to this controller, it will be activated, and only the ICourtBookingService will need to be resolved from the container. Only in cases where the UpcomingMaintenance method is the destination for the request will the ICourtMaintenanceService need to be resolved. Other patterns, such as the mediator pattern, are also popular ways to \ reduce the number of dependencies that a controller needs to have resolved. When you find yourself with controllers that have many dependencies, it may be time to consider reviewing if the actions should be separated into more focused controllers. Often this problem is indicative of a controller which has too many responsibilities.

Injecting Services into Middleware

Another common place to inject services is into ASP.NET Core middleware components. In this demo, we'll explore the two options we have for injecting services into middleware. We'll cover the different behavior of each option and learn why it's important to make the correct choice. While middleware components support constructor injection, the activation of middleware differs from the other framework components, such as controllers. This can have an important impact on the injection of dependencies. Here I have a middleware component called LastRequestMiddleware. This middleware is used to update the database with the last date and time of the request for authenticated users visiting the site. Currently, I'm injecting the UserManager into the constructor of this middleware. Let's run the application and see what happens. We get an InvalidOperationException. Focus on the error message, and you'll see that this exception is actually familiar. When we talked about scoped validation in the module called The Microsoft Dependency Injection Container, we saw that such an error suggests that we have incorrectly captured a scoped service inside a singleton. Let's jump back to the code and learn what has caused this exception. Middleware components are constructed once during the life of the application with any constructor dependencies being resolved from the root container. This means that the middleware components are essentially singletons within the application, and any dependencies injected for either constructor are therefore captured for the life of the application. This behavior differs when compared to controllers, for example, where we saw that a new controller instance is activated for every request. Therefore, when using constructor injection with middleware, we shouldn't inject short-lived scopes or transient services via the constructor. Here the UserManager dependency is a scoped service within the application. This is a good example of how the scope validation feature can be extremely useful because this would not have been an obvious problem otherwise. Without it, our application would've run and may have appeared to be working correctly. So how can we solve this? Middleware supports a second point of injection via the Invoke method or in asynchronous scenarios via the InvokeAsync method. This method is activated once per request, and its parameters are resolved from the container via the scope for the current request. To fix this application, I'll move the UserManager parameter into the InvokeAsync method. I can remove the private field as well. Finally, I'll change the code in the InvokeAsync method to use the method parameter. Let's run the application again. This time our application loads correctly without an exception and behaves as expected. When injecting dependencies into middleware, it is important to choose the correct point of injection to fit your requirements. Middleware constructors run once when the middleware is added to the application via the Configure method. Singleton services can be safely injected into the constructor, but all other lifetimes should be avoided. Scoped and transient services may end up being captured for longer than their intended lifetime if they are resolved via the constructor. The Invoke and InvokeAsync methods are called once per request. Services required by these methods are resolved from the request scope. Therefore, all lifetimes are supported for injection into these methods. There is one caveat for this guidance, which applies if you're using factory-based middleware. Factory-based middleware components implement the IMiddleware interface and are not activated via the conventional activation logic. Instead, the middleware are resolved when they're needed from the dependency injection container on a per-request basis. This allows transient and scoped services to be injected via their constructors. In my experience, the use of factory-based middleware is pretty rare.

Injecting Services into Razor Views

In ASP.NET Core, dependencies can also be injected directly into views. This includes traditional MVC Razor views, as well as the newer Razor pages. In this demo, we'll look at how we can inject services into Razor views. We'll then look at a specific use case where we can inject lookup data into the view, which supports populating the select options for a drop-down list. Here is the index Razor page, which displays the content for the site's home page. This section is responsible for rendering the weather information from the page model onto the UI. Currently, this is always included. However, the weather forecast is an optional feature of the application controlled via configuration. I'm going to add some code here to use the configuration to decide if this section should be rendered or not. To do that, I'm going to inject the configuration into the view. First, I'll add a using statement for the Configuration namespace within the application. Next, I can use the @inject directive to inject a service into this view. In this case, I'm requesting an IFeaturesConfiguration object be injected. This will be resolved from the dependency injection container and acts a bit like a property on the view. I'll name this injected dependency Features, which is the name that I can reference it by within the Razor code. Now that I have my configuration injected, I can use it to decide if the HTML for the weather forecast should be shown or not. This IA pretty straightforward way to consume services inside the view. I recommend against overusing view injection as it tends to mix concerns, and it can lead to business logic creeping into views where it doesn't belong. Cases where it is more reasonable to inject services directly is when they are specific to the concerns of rendering the view. An example of this is when populating lookup data for select options in a drop- down list. The available options for the drop-down are very much a concern of the UI and therefore the view. We can see an example of this in the AddStaffMember view used within the admin area of the site. The admin area uses traditional MVC controllers and views to render its content. In this view, I'm injecting an IStaffRolesOptionsService. This injected service is used to populate the drop-down list for the role selection by creating an IEnumerable of SelectListItems. The implementation for this is very simple, and it simply exposes a list of possible roles that staff members can perform at the tennis club. In this case, the data is hard-coded, but it could easily have been loaded on demand from a database if this service had had a database context injected. In the running application, we can see this drop-down list in use on the Add Staff Member's page in the admin area. The options for the list have been provided via an injected service into the view.

Creating and Using Scopes

In the Microsoft dependency injection container, scopes are used to manage the lifetime of objects. In ASP.NET Core, a scope is created automatically by the web host, which wraps each request. In this demo, we'll take a look at an example of manually creating a scope. We'll then use that scope to resolve services in the main method of the tennis booking application. For the vast majority of cases, you should never need to manually create a dependency injection scope in ASP.NET Core. There are some edge cases to this rule where it may be useful to access services outside of the normal request flow. And in those situations, you'll need to manually create and dispose of a scope. An example of one such exception is in situations where you need to run some application initialization, which depends on services registered with the container. Here in my main method of the Program class, I need to perform some one-time setup of the development database to ensure that two default users exist and have the correct roles. This process depends on the UserManager and RoleManager from ASP.NET Core Identity, both of which are registered into the container via the AddIdentityExtension method. To support this operation, I've split the calls which build and run the webhost into two stages. In the default templates, these would normally be chained together. After the webHost is built, I can now utilize its service provider, which is accessible via the Services property. This ServiceProvider will have been built using all of the services registered into the IServiceCollection in the ConfigureServices method. Next, I call the CreateScope method on the ServiceProvider, which will return me an IServiceScope. Since this implements IDisposable, I've placed this inside a using statement. A reasonable question at this point would be why do I need a scope at all? Why not simply request the service from the service provider directly? The answer is that the webhost service provider is the root application container. As a result, this container, and all objects it creates, will live for the life of the application. The UserManager and RoleManager escape services and depend on a scoped DbContext. If I were to resolve these directly from the root container, the instances would be essentially created as singletons and outlive their intended lifetime. This could introduce a memory leak into my application. Instead, by using a scope, I can ensure that their intended lifetime is respected. After creating the scope, I can request a service provider from it, which is bound to the scope. I can then use that service provider to resolve any dependencies needed to perform that application's initialization work. Here, for example, I retrieve the IHostingEnvironment by calling the GetRequiredService method. Further down, I use the ServiceProvider to resolve the UserManager and the RoleManager. Working with the database and applying migrations could be problematic if two instances of the application were to start at the same time, both attempting to migrate and seed the database. That is why in this example the code is limited to only seeding data inside the development environment. Once the initialization work is complete, the flow exits the using block, which ensures that the scope is now disposed of. As a result, scoped or transient services created within the scope will also be cleaned up and released for garbage collection. You should not need to manually create scopes very often inside ASP.NET Core applications since, by default, a scope is created for you per request, which supports the normal usage scenarios.

Review

In this module, we discussed the numerous places in ASP.NET Core applications where dependencies can be injected from a container. We discussed some of the rules relating to constructor injection and how a constructor will be chosen if multiple constructors exist. We learned about an alternative to constructor injection inside controllers using action injection and how this may be beneficial to application performance. We learned about the important difference between the two options for middleware injection due to the way the framework activates middleware. We saw an example of why we might choose to inject services into views and how to define view dependencies. And finally, we talked about how to create and use a manual scope from the ServiceProvider. Thank you for watching this module, and I hope to see you in the next one.

Beyond the Built-in Container

Introduction and Overview

In this module, we are going to discuss the options we have when the functionality of the built-in Microsoft container is simply not enough for some of the more advanced situations that we may need to handle. First, we'll look at extending the built-in container with two extra features available from a third party library called Scrutor. We'll look at using Scrutor to perform assembly scanning and to take advantage of the decorated pattern in our applications. When this still isn't enough, we'll look at the final option, which is to swap out the built-in container with one of the many third party alternatives.

Introducing and Installing Scrutor

Scrutor is a fantastic library, which fills two gaps in the service registration functionality available in the Microsoft container by providing support for assembly scanning and using the decorator pattern. Created by Kristian Hellang, the library is open source and available on GitHub if you want to explore the code or to raise issues. You can find it at github.com /khellang/Scrutor. It is conveniently available as a NuGet package, which you can reference in your application to begin using its features. The URL for the NuGet package is nuget.org /packages/Scrutor. I'm now going to add Scrutor to the tennis booking application. I'll do this via the NuGet Package Manager by right-clicking on the my project and choosing to manage NuGet packages. I'll make sure that I'm on the Browse tab so that I can search for Scrutor. The first entry is the package I need, so I'll select that. And I'll click the Install button to begin installing the library. I need to agree that the package can make changes to my solution by clicking OK to the dialog box, and that's it. If I edit my csproj file, we can see that Scrutor is now included as a package reference for this project.

Assembly Scanning with Scrutor

One of the features of Scrutor is providing assembly scanning support. In this demo, we'll see how we can utilize assembly scanning to reduce the number of manual service registrations in our code. Earlier, I applied rule pattern by registering multiple rule implementations against the ICourtBookingRule interface. Let's see how I can change my code to make use of assembly scanning to register my rules. In the BookingRuleServiceCollectionsExtensions class I created earlier, I have five rules being registered. I'll comment out the existing registration and apply Scrutor's assembly scanning here. Scrutor supports a fluent coding approach to scan for registrations, and I'll begin by using the ScanExtension method on the IServiceCollection. This takes an action on ITypeSourceSelector, which I'll refer to here as Scan. First I need to identify where Scrutor should scan for services. I can use the FromAssemblyOf method and pass a generic argument, which will act as a marker for the assembly to be scanned. Here I can use my ICourtBookingRule interface as all of my code is in a single assembly. Next I can choose what should be scanned for. In this case, I want to find all classes, which can be assigned to the ICourtBookingRule interface. I then need to define how the classes should be registered. My class's implement a single ICourtBookingRule interface, so I can require them to be registered as implementations of that interface, which will be the service type. Finally, I need to specify the service lifetime. This presents a slight problem for me since I've used both singleton and scoped registrations for my rules previously. For this first attempt, I'll be pragmatic and choose to register all of my rules as scoped. By choosing the lowest common lifetime, I can assure that the implementations which depend on scoped services will behave as expected. Since my rule classes are actually very simple, there's very little extra overhead in making them all scoped in this case. If I had a breakpoint after this registration and run the application, I can check the registrations have been scanned as expected. Here I can see that my five rules have been correctly registered by the scanning process. Each of them is registered with a scoped lifetime. The Scrutor assembly scanning code is only slightly shorter than the manual code that I had before. So what's the advantage of using assembly scanning here? For a few rules, manually registering them is not a problem. But as the complexity grows and more rules need to be added, it's entirely possible to forget to add the required registrations for them. Assembly scanning in this way means that I can add a new implementation of the ICourtBookingRule at any time in the future, and at runtime it will be found by the scanning process and registered for me. Ultimately, this reduces risk and manual work needed to maintain all of the registrations. I'm a little unhappy though with the fact that I'm now having to make all of my rules scoped. Is there anything I can do about that? There is one option, which will require a few small changes to my code. First, I'm going to head over to the ICourtBookingRule interface. I'll paste in two new derived interfaces, which will act as markers for the registrations in a moment. Next, I'll update each of my rules to implement the appropriate marker interface. For the services which I previously registered as singletons, I can update them to implement the ISingletonCourtBookingRule. And for scoped rules, I'll update them to implement IScopedCourtBookingRule. Finally, I'm ready to update the scanning configuration with Scrutor. I'll update the scan configuration to first look for all classes which implement the IScopedCourtBookingRule marker interface. I'll replace the call to AsImplementedInterfaces with a call to the As method, which takes a generic argument. This allows me to choose what service type the scanned registrations will end up being registered against. So here, I can force them to use the ICourtBookingRule as their registered service type. I now need to copy this to chain another scanning registration, this time scanning for classes marked with the ISingletonCourtBookingRule. I'll update the lifetime of this scanning registration to be singleton, which matches how I had these defined when I was manually registering them. I can now run the code again so that we can validate the registrations. I have my five rules registered as expected. And this time, they have the appropriate scoped or singleton lifetime as defined by their marker interfaces.

Applying the Decorator Pattern with Scrutor

The second feature which Scrutor adds to the Microsoft dependency injection container is support for applying the decorator pattern. The decorator pattern lets us wrap functionality of a service with additional functionality using another service, which conforms to a matching interface. This is a nice way to add features and extend functionality without modifying an existing class. In this demo, we'll see how we can apply the decorator pattern with Scrutor to decorate the weather service with caching to reduce the number of calls that we end up making to the remote API. As a reminder, here is the WeatherForecaster class in the tennis booking application. Every time this is called, it will make a call to the weather API via the WeatherApiClient, and then it transforms the response. This service is depended upon by the index PageModel for the home page. Since the home page is used very often, this could result in a lot of external API calls. So I'm going to add caching here to reduce the number of these calls that we need to make. One way to achieve this would be to add the caching code into the weather service itself. I'd like to keep the concern of caching and the concern of calling the API separate. I'll create a new class called CachedWeatherForecaster, which will implement the IWeatherForecaster interface. I'll implement the missing members, which I'll update in a moment. First, I'll add a constructor, which will accept an IWeatherForecaster. When using the decorator pattern, the service you are decorating or wrapping will be injected into the decorator. I'll also accept an IDistributedCache of CurrentWeatherResult. Now I can input by caching logic in the GetCurrentWeatherAsync method. I'll paste in some code here, and we can step through what it does. First, I define a cache key for the data, which includes the date of the weather forecast so that I ensure I don't return cache data for the wrong date. I can then check the cache using the TryGetValueAsync method. This returns a value tuple of a bool indicating if the key was found in the cache. If the key is found in the cache, then this method simply returns the data directly from the cache. Otherwise, I call the injected IWeatherForecaster, which is the service that this cache decorates. That returns its data by calling the API, and it gets added to the cache and then returns the caller of this method. I'll add two breakpoints to this code, one in the cached WeatherForecaster and one in the original WeatherForecaster. To wire all of this up, I can use Scrutor's decorator feature. Here is my existing registration for the IWeatherForecaster service. To wrap it with my cache implementation, I can call the Decorate method on the IServiceCollection. This accepts two generic arguments. The first is the service type that I'm decorating, and the second is the decorator class to use, the CachedWeatherForecaster in this case. If I run the application, the first time we hit the home page, the breakpoint in the cached service is hit. Since the weather data is not in the cache, if I continue, the breakpoint is hit in the WeatherForecaster class, and the API will be called. If I refresh the page, again the CachedWeatherForecaster breakpoint is hit. But this time, as the data is available in the cache, the WeatherForecaster itself is not called, and therefore an API call is never made. Due bear in mind that when using decorators, they're registered under the same lifetime as the original service. Here, for example, the original WeatherForecaster registration uses a singleton lifetime. As a result, the CachedWeatherForecaster registration will also use a singleton lifetime. That's perfectly appropriate here as my service doesn't depend on any scoped or transient service, but it's worth being aware of this to avoid your decorators accidentally capturing services under the wrong lifetimes. Applying a decorate pattern is a powerful way to wrap additional functionality around existing code without having to modify it. Scrutor adds basic decorator support for the Microsoft container, which should be sufficient for most common use cases.

Replacing the Built-in Dependency Injection Container

For many general situations, the Microsoft dependency injection container should be sufficient for registering and resolving services in your application. By design, it does lack some advanced container features, which you may find yourself needing to use, or you may prefer the registration syntax of another container. In this demo, we're going to take a short look at adding Autofac as the application dependency injection container in ASP.NET Core. This won't be a deep look at how to use Autofac itself. The focus here is to use it as an example of switching to a more feature-rich container. I'll start by installing the Autofac NuGet package for use in the tennis booking application. I'll go to Manage NuGet Packages and, again, ensure that I'm in the Browse section. I'll search for Autofac.extensions. The official package from Autofac here is called Autofac.Extensions .DependencyInjection. So I will select and install that one. I click OK to accept the two dialogs about modifying the project and accepting the license. Since the release of ASP.NET Core 1.1, there have been two ways that we can use a third party container in an ASP.NET Core application. The simplest option allows the third party container to be registered very early in the application pipeline as long as the container provides an implementation for the IServiceProvider factory interface. There are changes planned to the hosting model for ASP.NET Core 3.0, which will actually require this approach for all third party container integration. And therefore this is the option that I'm going to demonstrate. Fortunately, Autofac provides the required implementation of IServiceProviderFactory, so let's see how we can use it. In the program class, I can register Autofac with the WebHost builder. To do that, I'll add a call to the ConfigureServices method, which is an extension method on the IWebHost builder. This allows us to hook in very early to the hosting container in order to register services. In this case, I can use a lambda to call the AddAutofac extension method, which registers their implementation of the IServiceProviderFactory. I need to make sure that I have the appropriate using statement for Autofac.Extensions .DependencyInjection in this program class. To register services using Autofac, I can now use a new method in the Startup class called ConfigureContainer. This accepts as a parameter an Autofac ContainerBuilder. This method will be called by the framework after the normal ConfigureServices method has run. I can mix registrations, which is the IServiceCollection, in the ConfigureServices method with those using the container builder in the ConfigureContainer method. Repeat registrations in the ConfigureContainer method will supersede those registered in ConfigureServices. I'll create a simple scoped registration for the ICourtMaintenanceService using the Autofac builder. If I now comment out the code to the AddCourtServicesExtension method, which was previously responsible for registering the ICourtMaintenanceService, we can test that the Autofac registration has worked. If I run the application and navigate to the admin court maintenance page, which lists upcoming court maintenance, we can see that it still loads correctly. By design, the Microsoft dependency injection container lacks some advanced container features, which you may find yourself wanting or needing to use. As you've seen in this demo, we can easily swap out the Microsoft container with a more feature-rich option.

Review

In this module, we've looked at some of the options to extend the functionality of the built-in Microsoft container. We started by introducing and installing Scrutor, a third party library, which adds some extra extension methods for the IServiceCollection. We used Scrutor to add assembly scanning to the tennis booking application, as well as to apply the decorator pattern in order to add caching to the weather service. We concluded by looking at how we can bring in a more feature-rich third party container, such as Autofac, for situations where we need more advanced functionality. With that, we've reached the end of this course about dependency injection in ASP.NET Core. With the information that you've learned, you are ready to begin building complex ASP.NET Core applications, making full use of the built-in dependency injection container to register and resolve your services. Thank you very much for joining me in this course. I really hope that you found the information useful and that you've enjoyed watching.