CHAPTER 14

Using Dependency Injection

Services are objects that are shared between middleware components and endpoints. There are no restrictions on the features that services can provide, but they are usually used for tasks that are needed in multiple parts of the application, such as logging or database access.

The ASP.NET Core *dependency injection* feature is used to create and consume services. This is a topic that causes confusion and can be difficult to understand. In this chapter, I describe the problems that dependency injection solves and explain how dependency injection is supported by the ASP.NET Core platform. Table 14-1 puts dependency injection in context.

Table 14-1. *Putting Dependency Injection in Context*

Question	Answer
What is it?	Dependency Injection makes it easy to create loosely coupled components, which typically means that components consume functionality defined by interfaces without having any firsthand knowledge of which implementation classes are being used.
Why is it useful?	Dependency injection makes it easier to change the behavior of an application by changing the components that implement the interfaces that define application features. It also results in components that are easier to isolate for unit testing.
How is it used?	The `Startup` class is used to specify which implementation classes are used to deliver the functionality specified by the interfaces used by the application. Services can be explicitly requested through the `IServiceProvider` interface or by declaring constructor or method parameters.
Are there any pitfalls or limitations?	There are some differences in the way that middleware components and endpoints are handled and the way that services with different lifecycles are accessed.
Are there any alternatives?	You don't have to use dependency injection in your own code, but it is helpful to know how it works because it is used by the ASP.NET Core platform to provide features to developers.

Table 14-2 summarizes the chapter.

Table 14-2. *Chapter Summary*

Problem	Solution	Listing
Obtaining a service in the `Startup` class	Add a parameter to the `Configure` method	13
Obtaining a service in a middleware component	Define a constructor parameter	14, 33–35
Obtaining a service in an endpoint	Get an `IServiceProvider` object through the context objects	15–18
Instantiating a class that has constructor dependencies	Use the `ActivatorUtilities` class	19–21
Defining services that are instantiated for every dependency	Define transient services	22–27
Defining services that are instantiated for every request	Define scoped services	28–32
Accessing services in the `Startup.ConfigureServices` method	Define a `Startup` constructor parameter and assign the value to the property	36

A. Freeman, *Pro ASP.NET Core 3*, https://doi.org/10.1007/978-1-4842-5440-0_14

Table 14-2. (*continued*)

Problem	Solution	Listing
Managing service instantiation	Use a service factory	37, 38
Defining multiple implementations for a service	Define multiple services with the same scope and consume them through the `GetServices` method	39–41
Using services that support generic type parameters	Use a service with an unbound type	42

Preparing for This Chapter

In this chapter, I continue to use the Platform project from Chapter 13. New classes are required to prepare for this chapter. Start by creating the `Platform/Services` folder and add to it a class file named `IResponseFormatter.cs`, with the code shown in Listing 14-1.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from `https://github.com/apress/pro-asp.net-core-3`. See Chapter 1 for how to get help if you have problems running the examples.

Listing 14-1. The Contents of the IResponseFormatter.cs File in the Services Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace Platform.Services {
    public interface IResponseFormatter {

        Task Format(HttpContext context, string content);
    }
}
```

The `IResponseFormatter` interface defines a single method that receives an `HttpContext` object and a `string`. To create an implementation of the interface, add a class called `TextResponseFormatter.cs` to the `Platform/Services` folder with the code shown in Listing 14-2.

Listing 14-2. The Contents of the TextResponseFormatter.cs File in the Services Folder

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace Platform.Services {
    public class TextResponseFormatter : IResponseFormatter {
        private int responseCounter = 0;

        public async Task Format(HttpContext context, string content) {
            await context.Response.
                WriteAsync($"Response {++responseCounter}:\n{content}");
        }
    }
}
```

The `TextResponseFormatter` class implements the interface and writes the content to the response as a simple string with a prefix to make it obvious when the class is used.

Creating a Middleware Component and an Endpoint

Some of the examples in this chapter show how features are applied differently when using middleware and endpoints. Add a file called `WeatherMiddleware.cs` to the `Platform` folder with the code shown in Listing 14-3.

Listing 14-3. The Contents of the WeatherMiddleware.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace Platform {
    public class WeatherMiddleware {
        private RequestDelegate next;

        public WeatherMiddleware(RequestDelegate nextDelegate) {
            next = nextDelegate;
        }

        public async Task Invoke(HttpContext context) {
            if (context.Request.Path == "/middleware/class") {
                await context.Response
                    .WriteAsync("Middleware Class: It is raining in London");
            } else {
                await next(context);
            }
        }
    }
}
```

To create an endpoint that produces a similar result to the middleware component, add a file called `WeatherEndpoint.cs` to the `Platform` folder with the code shown in Listing 14-4.

Listing 14-4. The Contents of the WeatherEndpoint.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace Platform {
    public class WeatherEndpoint {

        public static async Task Endpoint(HttpContext context) {
            await context.Response
                .WriteAsync("Endpoint Class: It is cloudy in Milan");
        }
    }
}
```

Configuring the Request Pipeline

Replace the contents of the `Startup.cs` file with those shown in Listing 14-5. The classes defined in the previous section are applied alongside lambda functions that produce similar results.

Listing 14-5. Replacing the Contents of the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;
```

```
using Platform.Services;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();
            app.UseMiddleware<WeatherMiddleware>();

            IResponseFormatter formatter = new TextResponseFormatter();
            app.Use(async (context, next) => {
                if (context.Request.Path == "/middleware/function") {
                    await formatter.Format(context,
                        "Middleware Function: It is snowing in Chicago");
                } else {
                    await next();
                }
            });

            app.UseEndpoints(endpoints => {

                endpoints.MapGet("/endpoint/class", WeatherEndpoint.Endpoint);

                endpoints.MapGet("/endpoint/function", async context => {
                    await context.Response
                        .WriteAsync("Endpoint Function: It is sunny in LA");
                });
            });
        }
    }
}
```

Start the application by selecting Start Without Debugging or Run Without Debugging from the Debug menu or by opening a new PowerShell command prompt, navigating to the `Platform` project folder (which contains the `Platform.csproj` file, and running the command shown in Listing 14-6.

Listing 14-6. Starting the ASP.NET Core Runtime

```
dotnet run
```

Use a browser to request `http://localhost:5000/middleware/function`, and you will see the response shown in Figure 14-1. Each time you reload the browser, the counter shown in the response will be incremented.

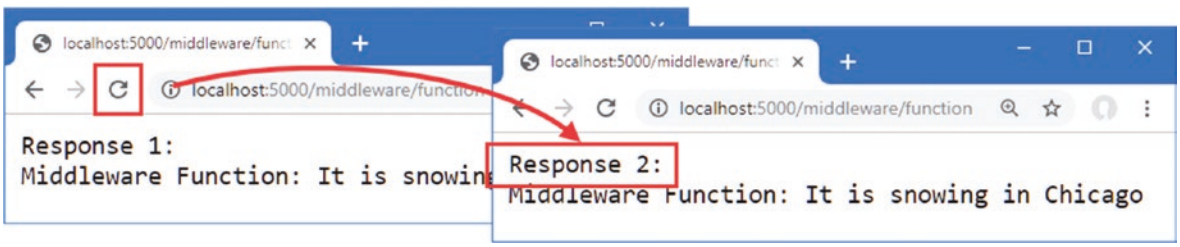

Figure 14-1. *Running the example application*

Understanding Service Location and Tight Coupling

To understand dependency injection, it is important to start with the two problems it solves. In the sections that follow, I describe both problems addressed by dependency injection.

TAKING A VIEW ON DEPENDENCY INJECTION

Dependency injection is one of the topics that readers contact me about most often. About half of the emails complain that I am "forcing" DI upon them. Oddly, the other half are complaints that I did not emphasize the benefits of DI strongly enough and other readers may not have realized how useful it can be.

Dependency injection can be a difficult topic to understand, and its value is contentious. DI can be a useful tool, but not everyone likes it—or needs it.

DI offers limited benefit if you are not doing unit testing or if you are working on a small, self-contained and stable project. It is still helpful to understand how DI works because DI is used to access some important ASP.NET Core features described in earlier chapters, but you don't always need to embrace DI in the custom classes you write. There are alternative ways of creating shared features—two of which I describe in the following sections—and using these is perfectly acceptable if you don't like DI.

I rely on DI in my own applications because I find that projects often go in unexpected directions, and being able to easily replace a component with a new implementation can save me a lot of tedious and error-prone changes. I'd rather put in some effort at the start of the project than do a complex set of edits later.

But I am not dogmatic about dependency injection and nor should you be. Dependency injection solves a problem that doesn't arise in every project, and only you can determine whether you need DI for your project.

Understanding the Service Location Problem

Most projects have features that need to be used in different parts of the application, which are known as *services*. Common examples include logging tools and configuration settings but can extend to any shared feature, including the `TextResponseFormatter` class that is defined in Listing 14-2 and used by one of the middleware components.

Each `TextResponseFormatter` object maintains a counter that is included in the response sent to the browser, and if I want to incorporate the same counter into the responses generated by other endpoints, I need to have a way to make a single `TextResponseFormatter` object available in such a way that it can be easily found and consumed at every point where responses are generated.

There are many ways to make services locatable, but there are two main approaches, aside from the one that is the main topic of this chapter. The first approach is to create an object and use it as a constructor or method argument to pass it to the part of the application where it is required. The other approach is to add a static property to the service class that provides direct access to the shared instance, as shown in Listing 14-7. This is known as the *singleton pattern*, and it was a common approach before the widespread use of dependency injection.

Listing 14-7. Creating a Singleton in the TextResponseFormatter.cs File in the Services Folder

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace Platform.Services {
    public class TextResponseFormatter : IResponseFormatter {
        private int responseCounter = 0;
        private static TextResponseFormatter shared;

        public async Task Format(HttpContext context, string content) {
            await context.Response.
                WriteAsync($"Response {++responseCounter}:\n{content}");
        }
```

```
        public static TextResponseFormatter Singleton {
            get {
                if (shared == null) {
                    shared = new TextResponseFormatter();
                }
                return shared;
            }
        }
    }
}
```

This is a basic implementation of the singleton pattern, and there are many variations that pay closer attention to issues such as safe concurrent access. What's important for this chapter is that the changes in Listing 14-7 rely on the consumers of the `TextResponseFormatter` service obtaining a shared object through the static `Singleton` property, as shown in Listing 14-8.

Listing 14-8. Using a Service in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;
using Platform.Services;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();
            app.UseMiddleware<WeatherMiddleware>();

            app.Use(async (context, next) => {
                if (context.Request.Path == "/middleware/function") {
                    await TextResponseFormatter.Singleton.Format(context,
                        "Middleware Function: It is snowing in Chicago");
                } else {
                    await next();
                }
            });

            app.UseEndpoints(endpoints => {

                endpoints.MapGet("/endpoint/class", WeatherEndpoint.Endpoint);

                endpoints.MapGet("/endpoint/function", async context => {
                    await TextResponseFormatter.Singleton.Format(context,
                        "Endpoint Function: It is sunny in LA");
                });
            });
        }
    }
}
```

The singleton pattern allows me to share a single `TextResponseFormatter` so it is used by a middleware component and an endpoint, with the effect that a single counter is incremented by requests for two different URLs. To see the effect of the singleton pattern, restart ASP.NET Core and request the `http://localhost:5000/middleware/function` and `http://localhost:5000/endpoint/function` URLs. A single counter is updated for both URLs, as shown in Figure 14-2.

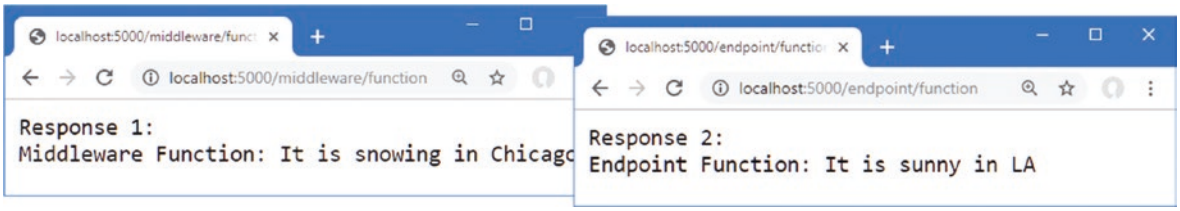

Figure 14-2. *Implementing the singleton pattern to create a shared service*

The singleton pattern is simple to understand and easy to use, but the knowledge of how services are located is spread throughout the application, and every service class and every service consumer needs to understand how to access the shared object. This can lead to variations in the singleton pattern as new services are created and creates many points in the code that must be updated when there is a change. This pattern can also be rigid and doesn't allow any flexibility in how services are managed because every consumer always shares a single service object.

Understanding the Tightly Coupled Components Problem

Although I defined an interface in Listing 14-1, the way that I have used the singleton pattern means that consumers are always aware of the implementation class they are using because that's the class whose static property is used to get the shared object. If I want to switch to a different implementation of the `IResponseFormatter` interface, I must locate every use of the service and replace the existing implementation class with the new one. There are patterns to solve this problem, too, such as the *type broker* pattern, in which a class provides access to singleton objects through their interfaces. Add a class file called `TypeBroker.cs` to the `Platform/Services` folder and use it to define the code shown in Listing 14-9.

Listing 14-9. The Contents of the TypeBroker.cs File in the Services Folder

```
namespace Platform.Services {
    public static class TypeBroker {
        private static IResponseFormatter formatter = new TextResponseFormatter();

        public static IResponseFormatter Formatter => formatter;
    }
}
```

The `Formatter` property provides access to a shared service object that implements the `IResponseFormatter` interface. Consumers of the service need to know that the `TypeBroker` class is responsible for selecting the implementation that will be used, but this pattern means that service consumers can work through interfaces rather than concrete classes, as shown in Listing 14-10.

Listing 14-10. Using a TypeBroker in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;
using Platform.Services;

namespace Platform {
    public class Startup {

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

```
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();
            app.UseMiddleware<WeatherMiddleware>();

            app.Use(async (context, next) => {
                if (context.Request.Path == "/middleware/function") {
                    await TypeBroker.Formatter.Format(context,
                        "Middleware Function: It is snowing in Chicago");
                } else {
                    await next();
                }
            });

            app.UseEndpoints(endpoints => {

                endpoints.MapGet("/endpoint/class", WeatherEndpoint.Endpoint);

                endpoints.MapGet("/endpoint/function", async context => {
                    await TypeBroker.Formatter.Format(context,
                        "Endpoint Function: It is sunny in LA");
                });
            });
        }
    }
}
```

This approach makes it easy to switch to a different implementation class by altering just the `TypeBroker` class and prevents service consumers from creating dependencies on a specific implementation. It also means that service classes can focus on the features they provide without having to deal with how those features will be located. To demonstrate, add a class file called `HtmlResponseFormatter.cs` to the `Platform/Services` folder with the code shown in Listing 14-11.

Listing 14-11. The Contents of the HtmlResponseFormatter.cs File in the Services Folder

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace Platform.Services {
    public class HtmlResponseFormatter : IResponseFormatter {

        public async Task Format(HttpContext context, string content) {
            context.Response.ContentType = "text/html";
            await context.Response.WriteAsync($@"
                <!DOCTYPE html>
                <html lang=""en"">
                <head><title>Response</title></head>
                <body>
                    <h2>Formatted Response</h2>
                    <span>{content}</span>
                </body>
                </html>");
        }
    }
}
```

This implementation of the `IResponseFormatter` sets the `ContentType` property of the `HttpResponse` object and inserts the content into an HTML template string. To use the new formatter class, I only need to change the `TypeBroker`, as shown in Listing 14-12.

Listing 14-12. Changing Implementation in the TypeBroker.cs File in the Platform/Services Folder

```
namespace Platform.Services {
    public static class TypeBroker {
        private static IResponseFormatter formatter = new HtmlResponseFormatter();

        public static IResponseFormatter Formatter => formatter;
    }
}
```

To make sure the new formatter works, restart ASP.NET Core and request `http://localhost:5000/endpoint/function`, which will produce the result shown in Figure 14-3.

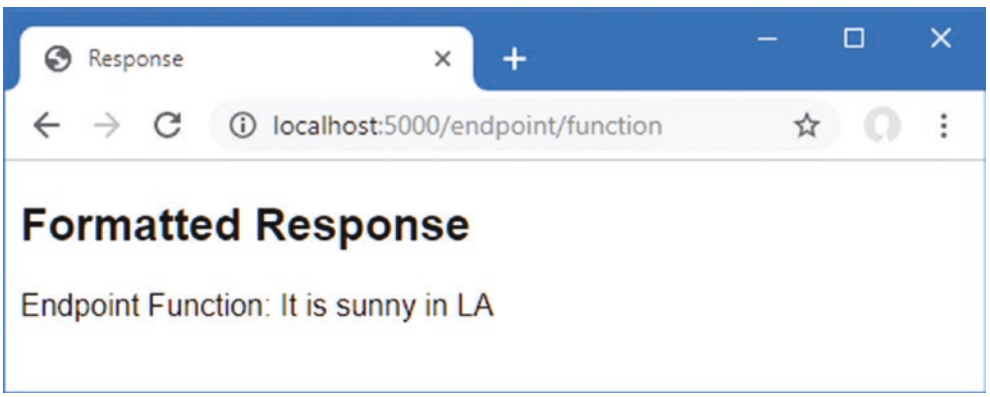

Figure 14-3. *Using a different service implementation class*

Using Dependency Injection

Dependency injection provides an alternative approach to providing services that tidy up the rough edges that arise in the singleton and type broker pattern, and it is integrated with other ASP.NET Core features. Listing 14-13 shows the use of ASP.NET Core dependency injection to replace the type broker from the previous section.

Listing 14-13. Using Dependency Injection in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;
using Platform.Services;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<IResponseFormatter, HtmlResponseFormatter>();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
                IResponseFormatter formatter) {

            app.UseDeveloperExceptionPage();
            app.UseRouting();
```

```
            app.UseMiddleware<WeatherMiddleware>();

            app.Use(async (context, next) => {
                if (context.Request.Path == "/middleware/function") {
                    await formatter.Format(context,
                        "Middleware Function: It is snowing in Chicago");
                } else {
                    await next();
                }
            });

            app.UseEndpoints(endpoints => {

                endpoints.MapGet("/endpoint/class", WeatherEndpoint.Endpoint);

                endpoints.MapGet("/endpoint/function", async context => {
                    await formatter.Format(context,
                        "Endpoint Function: It is sunny in LA");
                });
            });
        }
    }
}
```

Services are registered in the `ConfigureServices` method of the `Startup` class, using extensions methods on the `IServiceCollection` parameter. In Listing 14-13, I used an extension method to create a service for the `IResponseFormatter` interface.

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddSingleton<IResponseFormatter, HtmlResponseFormatter>();
}
...
```

The `AddSingleton` method is one of the extension methods available for services and tells ASP.NET Core that a single object should be used to satisfy demands for the service (the other extension methods are described in the "Using Service Lifecycles" section). The interface and the implementation class are specified as generic type arguments. To consume the service, I added a parameter to the `Configure` method.

```
...
public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
    IResponseFormatter formatter) {
...
```

The new parameter declares a dependency on the `IResponseFormatter` interface, and the method is said to depend on the interface. Before the `Configure` method is invoked, its parameters are inspected, the dependency is detected, and the application's services are inspected to determine whether it is possible to resolve the dependency. The statement in the `ConfigureServices` method tells the dependency injection system that a dependency on the `IResponseFormatter` interface can be resolved with an `HtmlResponseFormatter` object. The object is created and used as an argument to invoke the method. Because the object that resolves the dependency is provided from outside the class or function that uses it, it is said to have been injected, which is why the process is known as *dependency injection*.

Using a Service in a Middleware Class

Defining a service in the `ConfigureServices` method and consuming it in the `Configure` method may not seem impressive, but once a service is defined, it can be used almost anywhere in an ASP.NET Core application. Listing 14-14 declares a dependency on the `IResponseFormatter` interface in the middleware class defined at the start of the chapter.

Listing 14-14. Declaring a Dependency in the WeatherMiddleware.cs File in the Services Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;
using Platform.Services;

namespace Platform {
    public class WeatherMiddleware {
        private RequestDelegate next;
        private IResponseFormatter formatter;

        public WeatherMiddleware(RequestDelegate nextDelegate,
                IResponseFormatter respFormatter) {
            next = nextDelegate;
            formatter = respFormatter;
        }

        public async Task Invoke(HttpContext context) {
            if (context.Request.Path == "/middleware/class") {
                await formatter.Format(context,
                    "Middleware Class: It is raining in London");
            } else {
                await next(context);
            }
        }
    }
}
```

To declare the dependency, I added a constructor parameter. To see the result, restart ASP.NET Core and request the `http://localhost:5000/middleware/class` URL, which will produce the response shown in Figure 14-4.

Figure 14-4. *Declaring a dependency in a middleware class*

When the request pipeline is being set up, the ASP.NET Core platform reaches the statement in the `Configure` method that adds the `WeatherMiddleware` class as a component.

```
...
app.UseMiddleware<WeatherMiddleware>();
...
```

The platform understands it needs to create an instance of the WeatherMiddleware class and inspects the constructor. The dependency on the IResponseFormatter interface is detected, the services are inspected to see if the dependency can be resolved, and the shared service object is used when the constructor is invoked.

There are two important points to understand about this example. The first is that WeatherMiddleware doesn't know which implementation class will be used to resolve its dependency on the IResponseFormatter interface—it just knows that it will receive an object that conforms to the interface through its constructor parameter. Second, the WeatherMiddleware class doesn't know how the dependency is resolved—it just declares a constructor parameter. This is a more elegant approach than my implementations of the singleton and type broker patterns earlier in the chapter, and I can change the implementation class used to resolve the service by changing the generic type parameters used in the Startup.ConfigureServices method.

Using a Service in an Endpoint

The situation is more complicated in the WeatherEndpoint class, which is static and doesn't have a constructor through which dependencies can be declared. There are several approaches available to resolve dependencies for an endpoint class, which are described in the sections that follow.

Getting Services from the HttpContext Object

Services can be accessed through the HttpContext object that is received when a request is routed to an endpoint, as shown in Listing 14-15.

Listing 14-15. Using a Service in the WeatherEndpoint.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;
using Platform.Services;
using Microsoft.Extensions.DependencyInjection;

namespace Platform {
    public class WeatherEndpoint {

        public static async Task Endpoint(HttpContext context) {
            IResponseFormatter formatter =
                context.RequestServices.GetRequiredService<IResponseFormatter>();
            await formatter.Format(context, "Endpoint Class: It is cloudy in Milan");
        }
    }
}
```

The HttpContext.RequestServices property returns an object that implements the IServiceProvider interfaces, which provides access to the services that have been configured in the application's Start.ConfigureServices method. The Microsoft.Extensions.DependencyInjection namespace used in Listing 14-15 contains extension methods for the IServiceProvider interface that allow individual services to be obtained, as described in Table 14-3.

Table 14-3. *The IServiceProvider Extension Methods for Obtaining Services*

Name	Description
GetService<T>()	This method returns a service for the type specified by the generic type parameter or null if no such service has been defined.
GetService(type)	This method returns a service for the type specified or null if no such service has been defined.
GetRequiredService<T>()	This method returns a service specified by the generic type parameter and throws an exception if a service isn't available.
GetRequiredService(type)	This method returns a service for the type specified and throws an exception if a service isn't available.

When the Endpoint method is invoked, the GetRequiredService<T> method is used to obtain an IResponseFormatter object, which is used to format the response. To see the effect, restart ASP.NET Core and use the browser to request http://localhost:5000/endpoint/class, which will produce the formatted response shown in Figure 14-5.

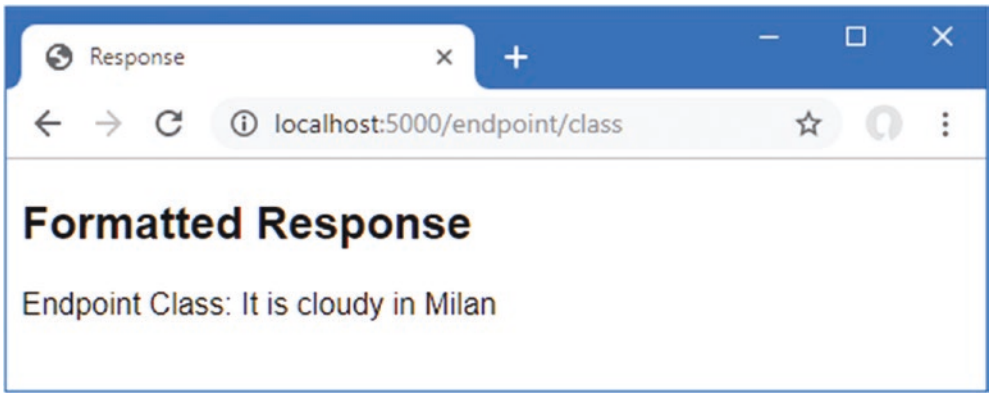

Figure 14-5. *Using a service in an endpoint class*

Using an Adapter Function

The drawback of using the HttpContext.RequestServices method is that the service must be resolved for every request that is routed to the endpoints. As you will learn later in the chapter, there are some services for which this is required because they provide features that are specific to a single request or response. This isn't the case for the IResponseFormatter service, where a single object can be used to format multiple responses.

A more elegant approach is to get the service when the endpoint's route is created and not for each request. Listing 14-16 changes the static Endpoint method so that it declares a dependency on the IResponseFormatter interface.

Listing 14-16. Defining an Adapter Function in the WeatherEndpoint.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;
using Platform.Services;

namespace Platform {
    public class WeatherEndpoint {

        public static async Task Endpoint(HttpContext context,
                IResponseFormatter formatter) {
            await formatter.Format(context, "Endpoint Class: It is cloudy in Milan");
        }
    }
}
```

Add a file called EndPointExtensions.cs to the Platform/Services folder and add the code shown in Listing 14-17.

Listing 14-17. The Contents of the EndpointExtensions.cs File in the Services Folder

```
using Microsoft.AspNetCore.Routing;
using Microsoft.Extensions.DependencyInjection;
using Platform.Services;

namespace Microsoft.AspNetCore.Builder {

    public static class EndpointExtensions {
        public static void MapWeather(this IEndpointRouteBuilder app, string path) {
```

```
            IResponseFormatter formatter =
                app.ServiceProvider.GetService<IResponseFormatter>();
            app.MapGet(path, context => Platform.WeatherEndpoint
                .Endpoint(context, formatter));
        }
    }
}
```

The new file creates an extension method for the `IEndpointRouterBuilder` interface, which is used to create routes in the `Startup` class. The interface defines a `ServiceProvider` property that returns an `IServiceProvider` object through which services can be obtained. The extension method gets the service and uses the `MapGet` method to register a `RequestDelegate` that passes on the `HttpContext` object and the `IResponseFormatter` object to the `WeatherEndpoint.Endpoint` method. In Listing 14-18, I have used the extension method to create the route for the endpoint.

Listing 14-18. Creating a Route in the Startup.cs File in the Platform Folder

```
...
app.UseEndpoints(endpoints => {

    //endpoints.MapGet("/endpoint/class", WeatherEndpoint.Endpoint);
    endpoints.MapWeather("/endpoint/class");

    endpoints.MapGet("/endpoint/function", async context => {
        await formatter.Format(context,
            "Endpoint Function: It is sunny in LA");
    });
});
...
```

The `MapWeather` extension method sets up the route and creates the adapter around the endpoint class. To see the result, restart ASP.NET Core and request the `http://localhost:5000/endpoint/class` URL, which will produce the same result as the previous example, shown in Figure 14-5.

Using the Activation Utility Class

I used static methods for endpoint classes in Chapter 13 because it makes them easier to use when creating routes. But for endpoints that require services, it can often be easier to use a class that can be instantiated because it allows for a more generalized approach to handling services. Listing 14-19 revises the endpoint with a constructor and removes the `static` keyword from the `Endpoint` method.

Listing 14-19. Revising the Endpoint in the WeatherEndpoint.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;
using Platform.Services;

namespace Platform {
    public class WeatherEndpoint {
        private IResponseFormatter formatter;

        public WeatherEndpoint(IResponseFormatter responseFormatter) {
            formatter = responseFormatter;
        }

        public async Task Endpoint(HttpContext context) {
            await formatter.Format(context, "Endpoint Class: It is cloudy in Milan");
        }
    }
}
```

The most common use of dependency injection in ASP.NET Core applications is in class constructors. Injection through methods, such as performed for middleware classes, is a complex process to re-create, but there are some useful built-in tools that take care of inspecting constructors and resolving dependencies using services, as shown in Listing 14-20.

Listing 14-20. Resolving Dependencies in the EndpointExtensions.cs File in the Services Folder

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;
using Microsoft.Extensions.DependencyInjection;
using Platform.Services;
using System.Reflection;
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Builder {

    public static class EndpointExtensions {

        public static void MapEndpoint<T> (this IEndpointRouteBuilder app,
                string path, string methodName = "Endpoint") {

            MethodInfo methodInfo = typeof(T).GetMethod(methodName);
            if (methodInfo == null || methodInfo.ReturnType != typeof(Task)) {
                throw new System.Exception("Method cannot be used");
            }
            T endpointInstance =
                ActivatorUtilities.CreateInstance<T>(app.ServiceProvider);
            app.MapGet(path,  (RequestDelegate)methodInfo
                .CreateDelegate(typeof(RequestDelegate), endpointInstance));
        }
    }
}
```

The extension method accepts a generic type parameter that specifies the endpoint class that will be used. The other arguments are the path that will be used to create the route and the name of the endpoint class method that processes requests.

A new instance of the endpoint class is created, and a delegate to the specified method is used to create a route. Like any code that uses .NET reflection, the extension method in Listing 14-20 can be difficult to read, but the key statement for the purposes of this chapter is this one:

```
...
T endpointInstance = ActivatorUtilities.CreateInstance<T>(app.ServiceProvider);
...
```

The `ActivatorUtilities` class, defined in the `Microsoft.Extensions.DependencyInjection` namespace, provides methods for instantiating classes that have dependencies declared through their constructor. Table 14-4 shows the most useful `ActivatorUtilities` methods.

Table 14-4. *The ActivatorUtilities Methods*

Name	Description
`CreateInstance<T>(services, args)`	This method creates a new instance of the class specified by the type parameter, resolving dependencies using the services and additional (optional) arguments.
`CreateInstance(services, type, args)`	This method creates a new instance of the class specified by the parameter, resolving dependencies using the services and additional (optional) arguments.
`GetServiceOrCreateInstance<T>(services, args)`	This method returns a service of the specified type, if one is available, or creates a new instance if there is no service.
`GetServiceOrCreateInstance(services, type, args)`	This method returns a service of the specified type, if one is available, or creates a new instance if there is no service.

Both methods resolve constructor dependencies using services through an `IServiceProvider` object and an optional array of arguments that are used for dependencies that are not services. These methods make it easy to apply dependency injection to custom classes, and the use of the `CreateInstance` method results in an extension method that can create routes with endpoint classes that consume services. Listing 14-21 uses the new extension method to create a route.

Listing 14-21. Creating a Route Using an Extension Method in the Startup.cs File in the Platform Folder

```
...
app.UseEndpoints(endpoints => {

    endpoints.MapEndpoint<WeatherEndpoint>("/endpoint/class");

    endpoints.MapGet("/endpoint/function", async context => {
        await formatter.Format(context,
            "Endpoint Function: It is sunny in LA");
    });
});
...
```

This type of extension method makes it easy to work with endpoint classes and provides a similar experience to the `UseMiddleware` method described in Chapter 12. To make sure that requests are routed to the endpoint, restart ASP.NET Core and request the `http://localhost:5000/endpoint/class` URL, which should produce the same response as shown in Figure 14-5.

Using Service Lifecycles

When I created the service in the previous section, I used the `AddSingleton` extension method, like this:

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddSingleton<IResponseFormatter, HtmlResponseFormatter>();
}
...
```

The `AddSingleton` method produces a service that is instantiated the first time it is used to resolve a dependency and is then reused for each subsequent dependency. This means that any dependency on the `IResponseFormatter` object will be resolved using the same `HtmlResponseFormatter` object.

Singletons are a good way to get started with services, but there are some problems for which they are not suited, so ASP.NET Core supports *scoped services*, which give a lifecycle for the objects that are created to resolve dependencies. Table 14-5 describes the set of methods used to create services. There are versions of these methods that accept types as conventional arguments, as demonstrated in the "Using Unbound Types in Services" section, later in this chapter.

There are versions of the methods in Table 14-5 that have a single type argument, which allows a service to be created that solves the service location problem without addressing the tightly coupled issue. You can see an example of this type of service in Chapter 24, where I share a simple data source that isn't accessed through an interface.

Table 14-5. *The Extension Methods for Creating Services*

Name	Description
`AddSingleton<T, U>()`	This method creates a single object of type `U` that is used to resolve all dependencies on type `T`.
`AddTransient<T, U>()`	This method creates a new object of type `U` to resolve each dependency on type `T`.
`AddScoped<T, U>()`	This method creates a new object of type `U` that is used to resolve dependencies on `T` within a single scope, such as request.

Creating Transient Services

The AddTransient method does the opposite of the AddSingleton method and creates a new instance of the implementation class for every dependency that is resolved. To create a service that will demonstrate the use of service lifecycles, add a file called GuidService.cs to the Platform/Services folder with the code shown in Listing 14-22.

Listing 14-22. The Contents of the GuidService.cs File in the Services Folder

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace Platform.Services {

    public class GuidService : IResponseFormatter {
        private Guid guid = Guid.NewGuid();

        public async Task Format(HttpContext context, string content) {
            await context.Response.WriteAsync($"Guid: {guid}\n{content}");
        }
    }
}
```

The Guid struct generates a unique identifier, which will make it obvious when a different instance is used to resolve a dependency on the IResponseFormatter interface. In Listing 14-23, I have changed the statement that creates the IResponseFormatter service to use the AddTransient method and the GuidService implementation class.

Listing 14-23. Creating a Transient Service in the Startup.cs File in the Platform Folder

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

If you restart ASP.NET Core and request the http://localhost:5000/endpoint/class and http://localhost:5000/middleware/class URLs, you will receive the responses shown in Figure 14-6. Each response will be shown with a different GUID value, confirming that transient service objects have been used to resolve the dependencies on the IResponseFormatter service for the endpoint and the middleware component. (The nature of GUIDs means you will see different values in your responses. What is important is that you don't see the same value used for both responses.)

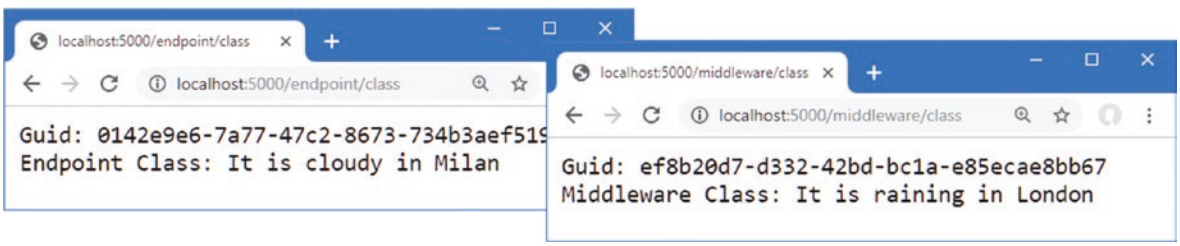

Figure 14-6. *Using transient services*

Avoiding the Transient Service Reuse Pitfall

The previous example demonstrated that when different service objects are created, the effect is not quite as you might expect, which you can see by clicking the reload buttons. Rather than seeing new GUID values, responses contain the same value, as shown in Figure 14-7.

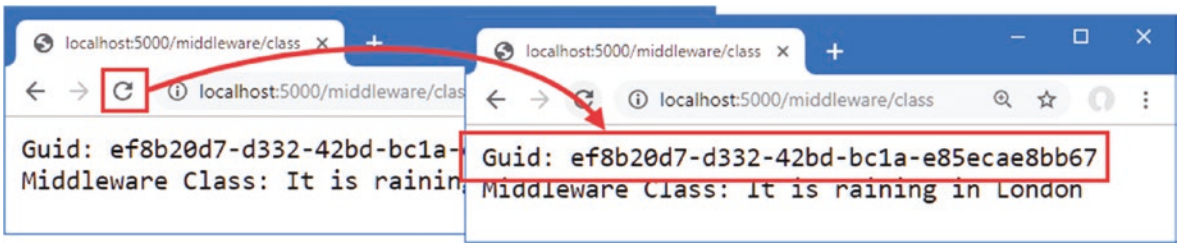

Figure 14-7. *The same GUID values appearing in responses*

New service objects are created only when dependencies are resolved, not when services are used. The components and endpoints in the example application have their dependencies resolved only when the application starts and the `Startup.Configure` method is invoked. Each receives a separate service object, which is then reused for every request that is processed.

To solve this problem for the middleware component, the dependency for the service can be moved to the `Invoke` method, as shown in Listing 14-24.

Listing 14-24. Moving a Dependency in the WeatherMiddleware.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;
using Platform.Services;

namespace Platform {
    public class WeatherMiddleware {
        private RequestDelegate next;
        //private IResponseFormatter formatter;

        public WeatherMiddleware(RequestDelegate nextDelegate) {
            next = nextDelegate;
            //formatter = respFormatter;
        }

        public async Task Invoke(HttpContext context, IResponseFormatter formatter) {
            if (context.Request.Path == "/middleware/class") {
                await formatter.Format(context,
                    "Middleware Class: It is raining in London");
            } else {
                await next(context);
            }
        }
    }
}
```

The ASP.NET Core platform will resolve dependencies declared by the `Invoke` method every time a request is processed, which ensures that a new transient service object is created.

The `ActivatorUtilities` class doesn't deal with resolving dependencies for methods, and ASP.NET Core includes this feature only for middleware components. The simplest way of solving this issue for endpoints is to explicitly request services when each request is handled, which is the approach I used earlier when showing how services are used. It is also possible to enhance the extension method to request services on behalf of an endpoint, as shown in Listing 14-25.

Listing 14-25. Requesting Services in the EndpointExtensions.cs File in the Services Folder

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;
using Microsoft.Extensions.DependencyInjection;
using Platform.Services;
```

```
using System.Reflection;
using System.Threading.Tasks;
using System.Linq;

namespace Microsoft.AspNetCore.Builder {

    public static class EndpointExtensions {

        public static void MapEndpoint<T>(this IEndpointRouteBuilder app,
                string path, string methodName = "Endpoint") {

            MethodInfo methodInfo = typeof(T).GetMethod(methodName);
            if (methodInfo == null || methodInfo.ReturnType != typeof(Task)) {
                throw new System.Exception("Method cannot be used");
            }

            T endpointInstance =
                ActivatorUtilities.CreateInstance<T>(app.ServiceProvider);

            ParameterInfo[] methodParams = methodInfo.GetParameters();
            app.MapGet(path, context => (Task)methodInfo.Invoke(endpointInstance,
                methodParams.Select(p => p.ParameterType == typeof(HttpContext)
                ? context
                : app.ServiceProvider.GetService(p.ParameterType)).ToArray()));
        }
    }
}
```

The code in Listing 14-25 isn't as efficient as the approach taken by the ASP.NET Core platform for middleware components. All the parameters defined by the method that handles requests are treated as services to be resolved, except for the `HttpContext` parameter. A route is created with a delegate that resolves the services for every request and invokes the method that handles the request. Listing 14-26 revises the `WeatherEndpoint` class to move the dependency on `IResponseFormatter` to the `Endpoint` method so that a new service object will be received for every request.

Listing 14-26. Moving the Dependency in the WeatherEndpoint.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;
using Platform.Services;

namespace Platform {
    public class WeatherEndpoint {
        //private IResponseFormatter formatter;

        //public WeatherEndpoint(IResponseFormatter responseFormatter) {
        //    formatter = responseFormatter;
        //}

        public async Task Endpoint(HttpContext context,
                IResponseFormatter formatter) {
            await formatter.Format(context, "Endpoint Class: It is cloudy in Milan");
        }
    }
}
```

The changes in Listing 14-24 to Listing 14-26 ensure that the transient service is resolved for every request, which means that a new `GuidService` object is created and every response contains a unique ID.

For the middleware and endpoint defined as lambda expressions, the service must be obtained as each request is handled because the dependency declared by the Configure method parameter is resolved only once, when the request pipeline is configured. Listing 14-27 shows the changes required to get a new service object.

Listing 14-27. Using a Transient Service in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;
using Platform.Services;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddTransient<IResponseFormatter, GuidService>();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();
            app.UseMiddleware<WeatherMiddleware>();

            app.Use(async (context, next) => {
                if (context.Request.Path == "/middleware/function") {
                    IResponseFormatter formatter
                        = app.ApplicationServices.GetService<IResponseFormatter>();
                    await formatter.Format(context,
                        "Middleware Function: It is snowing in Chicago");
                } else {
                    await next();
                }
            });

            app.UseEndpoints(endpoints => {

                endpoints.MapEndpoint<WeatherEndpoint>("/endpoint/class");

                endpoints.MapGet("/endpoint/function", async context => {
                    IResponseFormatter formatter
                        = app.ApplicationServices.GetService<IResponseFormatter>();
                    await formatter.Format(context,
                        "Endpoint Function: It is sunny in LA");
                });
            });
        }
    }
}
```

Restart ASP.NET Core, navigate to any of the four URLs supported by the application (`http://localhost:5000/middleware/class`, `/middleware/function`, `/endpoint/class`, and `/endpoint/function`), and click the browser's reload button. Each time you reload, a new request is sent to ASP.NET Core, and the component or endpoint that handles the request receives a new service object, such that a different GUID is shown in each response, as shown in Figure 14-8.

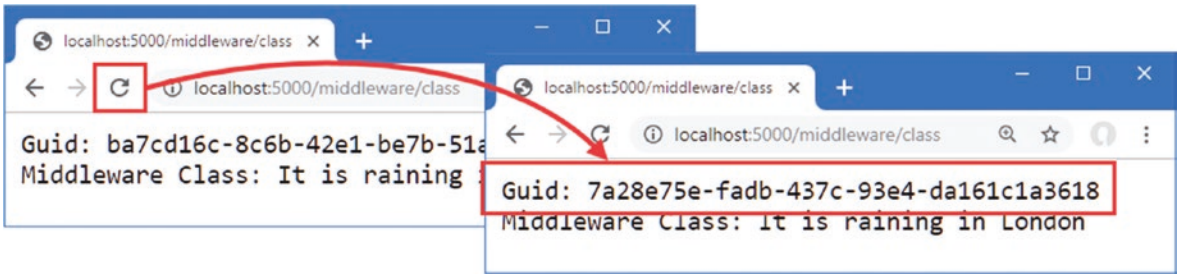

Figure 14-8. *Using a transient service*

Using Scoped Services

Scoped services strike a balance between singleton and transient services. Within a scope, dependencies are resolved with the same object. A new scope is started for each HTTP request, which means that a service object will be shared by all the components that handle that request. To prepare for a scoped service, Listing 14-28 changes the `WeatherMiddleware` class to declare three dependencies on the same service.

Listing 14-28. Adding Dependencies in the WeatherMiddleware.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;
using Platform.Services;

namespace Platform {
    public class WeatherMiddleware {
        private RequestDelegate next;

        public WeatherMiddleware(RequestDelegate nextDelegate) {
            next = nextDelegate;
        }

        public async Task Invoke(HttpContext context, IResponseFormatter formatter1,
                IResponseFormatter formatter2, IResponseFormatter formatter3) {
            if (context.Request.Path == "/middleware/class") {
                await formatter1.Format(context, string.Empty);
                await formatter2.Format(context, string.Empty);
                await formatter3.Format(context, string.Empty);
            } else {
                await next(context);
            }
        }
    }
}
```

Declaring several dependencies on the same service isn't required in real projects, but it is useful for this example because each dependency is resolved independently. Since the `IResponseFormatter` service was created with the `AddTransient` method, each dependency is resolved with a different object. Restart ASP.NET Core and request `http://localhost:5000/middleware/class`, and you will see that a different GUID is used for each of the three messages written to the response, as shown in Figure 14-9. When you reload the browser, a new set of three GUIDs is displayed.

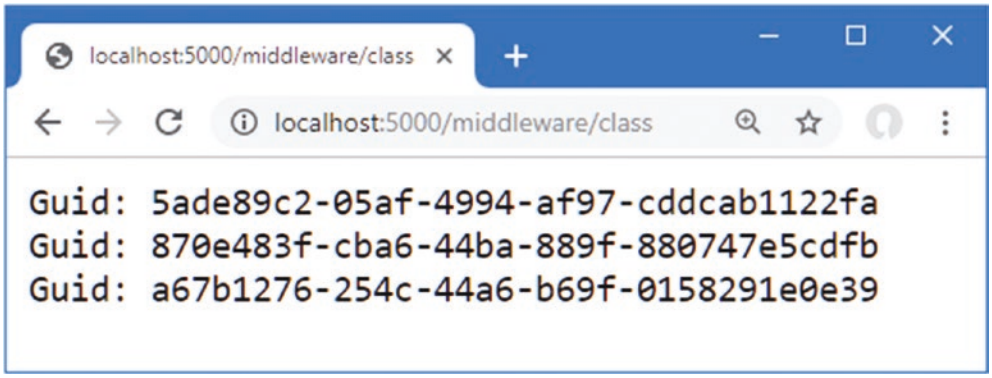

Figure 14-9. *Resolving dependencies on a transient service*

Listing 14-29 changes the `IResponseFormatter` service to use the scoped lifecycle with the `AddScoped` method.

■ **Tip** You can create your own scopes through the `CreateScope` extension method for the `IServiceProvider` interface. The result is an `IServiceProvider` that is associated with a new scope and that will have its own implementation objects for scoped services.

Listing 14-29. Using a Scoped Service in the Startup.cs File in the Platform Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddScoped<IResponseFormatter, GuidService>();
}
...
```

Restart ASP.NET Core and request `http://localhost:5000/middleware/class` again, and you will see that the same GUID is used to resolve all three dependencies declared by the middleware component, as shown in Figure 14-10. When the browser is reloaded, the HTTP request sent to ASP.NET Core creates a new scope and a new service object.

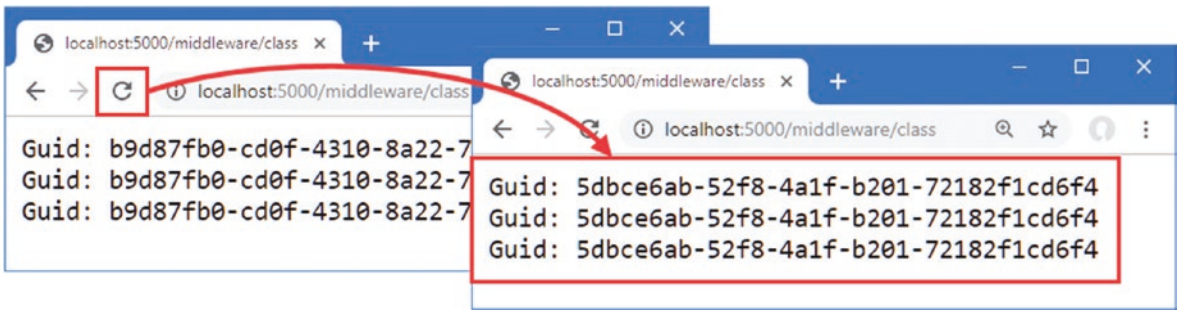

Figure 14-10. *Using a scoped service*

Avoiding the Scoped Service Validation Pitfall

Service consumers are unaware of the lifecycle that has been selected for singleton and transient services: they declare a dependency or request a service and get the object they require.

Scoped services can be used only within a scope. A new scope is created automatically for each request that was received. Requesting a scoped service outside of a scope causes an exception. To see the problem, request `http://localhost:5000/endpoint/class`, which will generate the exception response shown in Figure 14-11.

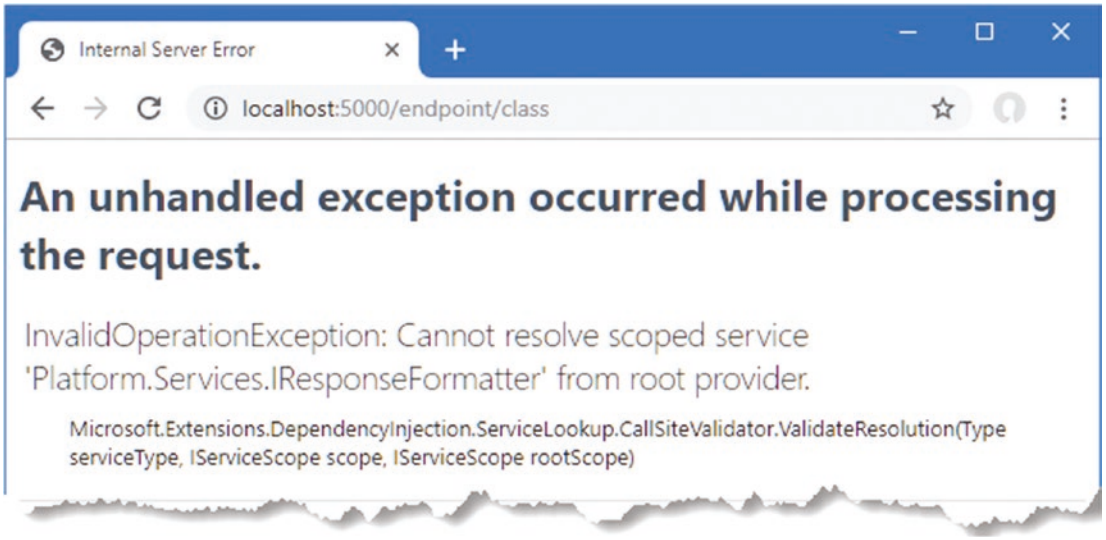

Figure 14-11. *Requesting a scoped service*

The extension method that configures the endpoint resolves services through an `IServiceProvider` object obtained from the routing middleware, like this:

```
...
app.MapGet(path, context => (Task)methodInfo.Invoke(endpointInstance,
    methodParams.Select(p => p.ParameterType == typeof(HttpContext)
    ? context : app.ServiceProvider.GetService(p.ParameterType)).ToArray()));
...
```

This is known as the *root provider*, and it does not provide access to scoped services to prevent the accidental use of services that are not intended for use outside of a scope.

Accessing Scoped Services Through the Context Object

The `HttpContext` class defines a `RequestServices` property that returns an `IServiceProvider` object that allows access to scoped services, as well as singleton and transient services. This fits well with the most common use of scoped services, which is to use a single service object for each HTTP request. Listing 14-30 revises the endpoint extension method so that dependencies are resolved using the services provided through the `HttpContext`.

Listing 14-30. Using a Scoped Service in the EndpointExtensions.cs File in the Services Folder

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;
using Microsoft.Extensions.DependencyInjection;
using Platform.Services;
using System.Reflection;
using System.Threading.Tasks;
using System.Linq;

namespace Microsoft.AspNetCore.Builder {

    public static class EndpointExtensions {

        public static void MapEndpoint<T>(this IEndpointRouteBuilder app,
                string path, string methodName = "Endpoint") {
```

```
            MethodInfo methodInfo = typeof(T).GetMethod(methodName);
            if (methodInfo == null || methodInfo.ReturnType != typeof(Task)) {
                throw new System.Exception("Method cannot be used");
            }

            T endpointInstance =
                ActivatorUtilities.CreateInstance<T>(app.ServiceProvider);

            ParameterInfo[] methodParams = methodInfo.GetParameters();
            app.MapGet(path, context => (Task)methodInfo.Invoke(endpointInstance,
                methodParams.Select(p => p.ParameterType == typeof(HttpContext)
                ? context
                : context.RequestServices.GetService(p.ParameterType)).ToArray()));
        }
    }
}
```

Only dependencies that are declared by the method that handles the request are resolved using the `HttpContext.RequestServices` property. Services that are declared by an endpoint class constructor are still resolved using the `IEndpointRouteBuilder.ServiceProvider` property, which ensures that endpoints don't use scoped services inappropriately.

Creating New Handlers for Each Request

The problem with the extension method is that it requires endpoint classes to know the lifecycles for the services they depend on. The `WeatherEndpoint` class depends on the `IResponseFormatter` service and must know that a dependency can be declared only through the `Endpoint` method and not the constructor.

To remove the need for this knowledge, a new instance of the endpoint class can be created to handle each request, as shown in Listing 14-31, which allows constructor and method dependencies to be resolved without needing to know which services are scoped.

Listing 14-31. Instantiating Endpoints in the EndpointExtensions.cs File in the Services Folder

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;
using Microsoft.Extensions.DependencyInjection;
using Platform.Services;
using System.Reflection;
using System.Threading.Tasks;
using System.Linq;

namespace Microsoft.AspNetCore.Builder {

    public static class EndpointExtensions {

        public static void MapEndpoint<T>(this IEndpointRouteBuilder app,
                string path, string methodName = "Endpoint") {

            MethodInfo methodInfo = typeof(T).GetMethod(methodName);
            if (methodInfo == null || methodInfo.ReturnType != typeof(Task)) {
                throw new System.Exception("Method cannot be used");
            }

            ParameterInfo[] methodParams = methodInfo.GetParameters();
            app.MapGet(path, context => {
                T endpointInstance =
                    ActivatorUtilities.CreateInstance<T>(context.RequestServices);
```

```
                return (Task)methodInfo.Invoke(endpointInstance,
                    methodParams.Select(p =>
                     p.ParameterType == typeof(HttpContext) ? context :
                     context.RequestServices.GetService(p.ParameterType)).ToArray());
            });
        }
    }
}
```

This approach requires a new instance of the endpoint class to handle each request, but it ensures that no knowledge of service lifecycles is required.

Using Scoped Services in Lambda Expressions

The `HttpContext` class can also be used in middleware components and endpoints that are defined as lambda expressions, as shown in Listing 14-32.

Listing 14-32. Using a Scoped Service in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;
using Platform.Services;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddScoped<IResponseFormatter, GuidService>();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();
            app.UseMiddleware<WeatherMiddleware>();

            app.Use(async (context, next) => {
                if (context.Request.Path == "/middleware/function") {
                    IResponseFormatter formatter
                        = context.RequestServices.GetService<IResponseFormatter>();
                    await formatter.Format(context,
                        "Middleware Function: It is snowing in Chicago");
                } else {
                    await next();
                }
            });

            app.UseEndpoints(endpoints => {

                endpoints.MapEndpoint<WeatherEndpoint>("/endpoint/class");

                endpoints.MapGet("/endpoint/function", async context => {
                    IResponseFormatter formatter
                        = context.RequestServices.GetService<IResponseFormatter>();
                    await formatter.Format(context,
                        "Endpoint Function: It is sunny in LA");
```

```
                });
            });
        }
    }
}
```

Restart ASP.NET Core and request the URLs that target the lambda functions: `http://localhost:5000/endpoint/function` and `/middleware/function`. The scoped service will be obtained without an exception, producing the responses shown in Figure 14-12.

Figure 14-12. *Using scoped services in lambda functions*

Other Dependency Injection Features

In the sections that follow, I describe some additional features available when using dependency injection. These are not required for all projects, but they are worth understanding because they provide context for how dependency injection works and can be helpful when the standard features are not quite what a project requires.

Creating Dependency Chains

When a class is instantiated to resolve a service dependency, its constructor is inspected, and any dependencies on services are resolved. This allows one service to declare a dependency on another service, creating a chain that is resolved automatically. To demonstrate, add a class file called `TimeStamping.cs` to the `Platform/Services` folder with the code shown in Listing 14-33.

Listing 14-33. The Contents of the TimeStamping.cs File in the Services Folder

```
using System;

namespace Platform.Services {

    public interface ITimeStamper {
        string TimeStamp { get; }
    }

    public class DefaultTimeStamper : ITimeStamper {

        public string TimeStamp {
            get => DateTime.Now.ToShortTimeString();
        }
    }
}
```

The class file defines an interface named `ITimeStamper` and an implementation class named `DefaultTimeStamper`. Next, add a file called `TimeResponseFormatter.cs` to the `Platform/Services` folder with the code shown in Listing 14-34.

Listing 14-34. The Contents of the TimeResponseFormatter.cs File in the Services Folder

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace Platform.Services {
    public class TimeResponseFormatter : IResponseFormatter {
        private ITimeStamper stamper;

        public TimeResponseFormatter(ITimeStamper timeStamper) {
            stamper = timeStamper;
        }

        public async Task Format(HttpContext context, string content) {
            await context.Response.WriteAsync($"{stamper.TimeStamp}: {content}");
        }
    }
}
```

The `TimeResponseFormatter` class is an implementation of the `IResponseFormatter` interface that declares a dependency on the `ITimeStamper` interface with a constructor parameter. Listing 14-35 defines services for both interfaces in the `ConfigureServices` method of the `Startup` class.

■ **Note** Services don't need to have the same lifecycle as their dependencies, but you can end up with odd effects if you mix lifecycles. Lifecycles are applied only when a dependency is resolved, which means that if a scoped service depends on a transient service, for example, then the transient object will behave as though it was assigned the scoped lifecycle.

Listing 14-35. Configuring Services in the Startup.cs File in the Platform Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddScoped<IResponseFormatter, TimeResponseFormatter>();
    services.AddScoped<ITimeStamper, DefaultTimeStamper>();
}
...
```

When a dependency on the `IResponseFormatter` service is resolved, the `TimeResponseFormatter` constructor will be inspected, and its dependency on the `ITimeStamper` service will be detected. A `DefaultTimeStamper` object will be created and injected into the `TimeResponseFormatter` constructor, which allows the original dependency to be resolved. To see the dependency chain in action, restart ASP.NET Core and request `http://localhost:5000/middleware/function`, and you will see the timestamp generated by the `DefaultTimeStamper` class included in the response produced by the `TimeResponseFormatter` class, as shown in Figure 14-13.

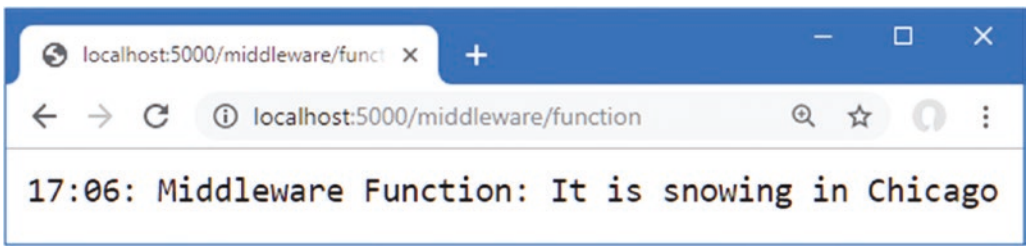

Figure 14-13. *Creating a chain of dependencies*

Accessing Services in the ConfigureServices Method

The dependency injection feature is set up by the Platform class before the Startup class is instantiated and the ConfigureServices method is invoked. During the setup process, the services required by the ASP.NET Core platform are created, as well as the basic services provided to applications, such as configuration data, logging, and the environment setting, all of which are described in earlier chapters.

The Startup class can declare dependencies on these services by defining a constructor. When the Startup class is instantiated during application startup, the constructor is inspected, and the dependencies it declares are resolved. The objects injected into the constructor can be assigned to properties and accessed in the ConfigureService method, which adds the services that are specific to the application. The combined set of services is then available to the Configure method and to the middleware components and endpoints that process requests.

■ **Note** The methods defined by the Startup class are not treated the same way. The Configure method can declare dependencies using parameters, but you will receive an exception if you add parameters to the ConfigureServices method. The technique described in this section is the only way to declare dependencies on services for use in the ConfigureServices method.

The most common use of the Startup constructor is to declare a dependency on the IConfiguration service, which provides access to the application's configuration data, as shown in Listing 14-36. The configuration data is described in detail in Chapter 15.

Listing 14-36. Declaring a Dependency in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Routing;
using Microsoft.Extensions.DependencyInjection;
using Platform.Services;
using Microsoft.Extensions.Configuration;

namespace Platform {

    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration;

        public void ConfigureServices(IServiceCollection services) {
            services.AddScoped<IResponseFormatter, TimeResponseFormatter>();
            services.AddScoped<ITimeStamper, DefaultTimeStamper>();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

            // ...statements omitted for brevity...
        }
    }
}
```

The IConfiguration service is received through the constructor and assigned to a property named Configuration, which can then be used by the ConfigureServices method. This example doesn't change the responses produced by the application, but you can see how configuration data can be used to configure services in the next section.

Using Service Factory Functions

Factory functions allow you to take control of the way that service implementation objects are created, rather than relying on ASP.NET Core to create instances for you. There are factory versions of the `AddSingleton`, `AddTransient`, and `AddScoped` methods, all of which are used with a function that receives an `IServiceProvider` object and returns an implementation object for the service.

One use for factory functions is to define the implementation class for a service as a configuration setting, which is read through the `IConfguration` service. This requires the pattern described in the previous section so that the configuration data is accessible in the `ConfigureServices` method. Listing 14-37 adds a factory function for the `IResponseFormatter` service that gets the implementation class from the configuration data.

Listing 14-37. Using a Factory Function in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;
using Platform.Services;
using Microsoft.Extensions.Configuration;
using System;

namespace Platform {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration;

        public void ConfigureServices(IServiceCollection services) {
            services.AddScoped<IResponseFormatter>(serviceProvider => {
                string typeName = Configuration["services:IResponseFormatter"];
                return (IResponseFormatter)ActivatorUtilities
                    .CreateInstance(serviceProvider, typeName == null
                        ? typeof(GuidService) : Type.GetType(typeName, true));
            });
            services.AddScoped<ITimeStamper, DefaultTimeStamper>();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

            // ...statements omitted for brevity...
        }
    }
}
```

The factory function reads a value from the configuration data, which is converted into a type and passed to the `ActivatorUtilities.CreateInstance` method. Listing 14-38 adds a configuration setting to the `appsettings.Development.json` file that selects the `HtmlResponseFormatter` class as the implementation for the `IResponseFormatter` service. The JSON configuration file is described in detail in Chapter 15.

Listing 14-38. Defining a Setting in the appsettings.Development.json File in the Platform Folder

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "services": {
    "IResponseFormatter": "Platform.Services.HtmlResponseFormatter"
  }
}
```

When a dependency on the `IResponseFormatter` service is resolved, the factory function creates an instance of the type specified in the configuration file. Restart ASP.NET Core and request the `http://localhost:5000/middleware/function` URL, which will produce the response shown in Figure 14-14.

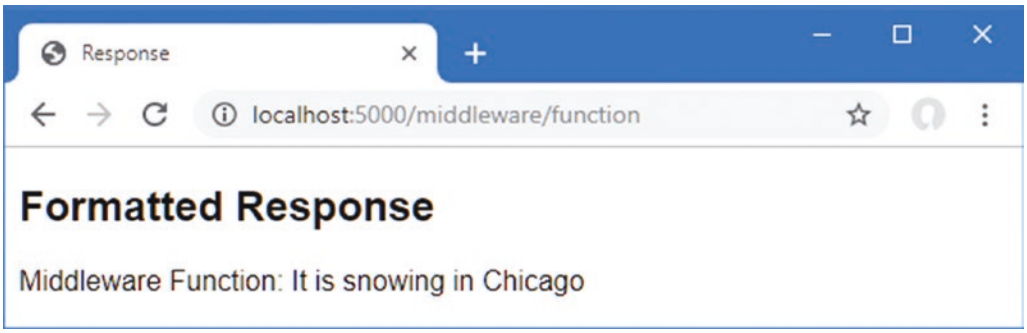

Figure 14-14. *Using a service factory*

Creating Services with Multiple Implementations

Services can be defined with multiple implementations, which allows a consumer to select an implementation that best suits a specific problem. This is a feature that works best when the service interface provides insight into the capabilities of each implementation class. To provide information about the capabilities of the `IResponseFormatter` implementation classes, add the default property shown in Listing 14-39 to the interface.

Listing 14-39. Adding a Property in the IResponseFormatter.cs File in the Services Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace Platform.Services {
    public interface IResponseFormatter {

        Task Format(HttpContext context, string content);

        public bool RichOutput => false;
    }
}
```

This `RichOutput` property will be `false` for implementation classes that don't override the default value. To ensure there is one implementation that returns `true`, add the property shown in Listing 14-40 to the `HtmlResponseFormatter` class.

Listing 14-40. Overriding a Property in the HtmlResponseFormatter.cs File in the Services Folder

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace Platform.Services {
    public class HtmlResponseFormatter : IResponseFormatter {

        public async Task Format(HttpContext context, string content) {
            context.Response.ContentType = "text/html";
            await context.Response.WriteAsync($@"
                <!DOCTYPE html>
                <html lang=""en"">
                <head><title>Response</title></head>
                <body>
                    <h2>Formatted Response</h2>
                    <span>{content}</span>
                </body>
                </html>");
        }

        public bool RichOutput => true;
    }
}
```

Listing 14-41 registers multiple implementations for the `IResponseFormatter` service, which is done by making repeated calls to the `Add<lifecycle>` method. The listing also replaces the existing request pipeline with two routes that demonstrate how the service can be used.

Listing 14-41. Defining and Using a Service in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;
using Platform.Services;
using Microsoft.Extensions.Configuration;
using System;
using System.Linq;

namespace Platform {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration;

        public void ConfigureServices(IServiceCollection services) {
            services.AddScoped<ITimeStamper, DefaultTimeStamper>();
            services.AddScoped<IResponseFormatter, TextResponseFormatter>();
            services.AddScoped<IResponseFormatter, HtmlResponseFormatter>();
            services.AddScoped<IResponseFormatter, GuidService>();
        }
```

```
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapGet("/single", async context => {
                    IResponseFormatter formatter = context.RequestServices
                        .GetService<IResponseFormatter>();
                    await formatter.Format(context, "Single service");
                });

                endpoints.MapGet("/", async context => {
                    IResponseFormatter formatter = context.RequestServices
                        .GetServices<IResponseFormatter>().First(f => f.RichOutput);
                    await formatter.Format(context, "Multiple services");
                });
            });
        }
    }
}
```

The AddScoped statements register three services for the IResponseFormatter interface, each with a different implementation class. The route for the /single URL uses the IServiceProvider.GetService<T> method to request a service, like this:

```
...
context.RequestServices.GetService<IResponseFormatter>();
...
```

This is a service consumer that is unaware that there are multiple implementations available. The service is resolved using the most recently registered implementation, which is the GuidService class. Restart ASP.NET Core and request http://localhost:5000/single, and you will see the output on the left side of Figure 14-15.

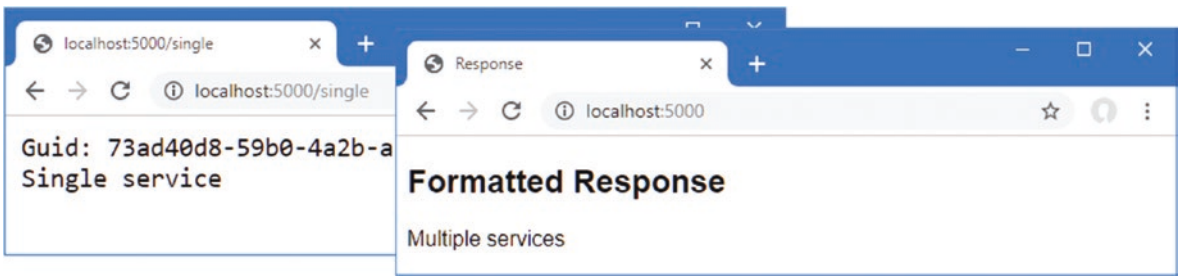

Figure 14-15. *Using multiple service implementations*

The other endpoint is a service consumer that is aware that multiple implementations may be available and that requests the service using the IServiceProvider.GetServices<T> method.

```
...
context.RequestServices.GetServices<IResponseFormatter>().First(f => f.RichOutput);
...
```

This method returns an IEnumerable<IResponseFormatter> that enumerates the implementations that are available. These are filtered using the LINQ First method to select an implementation whose RichOutput property returns true. If you request http://localhost:5000, you will see the output on the right of Figure 14-15, showing that the endpoint has selected the service implementation that best suits its needs.

Using Unbound Types in Services

Services can be defined with generic type parameters that are bound to specific types when the service is requested, as shown in Listing 14-42.

Listing 14-42. Using an Unbound Type in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;
using Platform.Services;
using Microsoft.Extensions.Configuration;
using System;
using System.Linq;
using System.Collections.Generic;

namespace Platform {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration;

        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton(typeof(ICollection<>), typeof(List<>));
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapGet("/string", async context => {
                    ICollection<string> collection
                        = context.RequestServices.GetService<ICollection<string>>();
                    collection.Add($"Request: { DateTime.Now.ToLongTimeString() }");
                    foreach (string str in collection) {
                        await context.Response.WriteAsync($"String: {str}\n");
                    }
                });

                endpoints.MapGet("/int", async context => {
                    ICollection<int> collection
                        = context.RequestServices.GetService<ICollection<int>>();
                    collection.Add(collection.Count() + 1);
                    foreach (int val in collection) {
                        await context.Response.WriteAsync($"Int: {val}\n");
                    }
                });
            });
        }
    }
}
```

This feature relies on the versions of the AddSIngleton, AddScoped, and AddTransient methods that accept types as conventional arguments and cannot be performed using generic type arguments. The service in Listing 14-42 is created with unbound types, like this:

```
...
services.AddSingleton(typeof(ICollection<>), typeof(List<>));
...
```

When a dependency on an ICollection<T> service is resolved, a List<T> object will be created so that a dependency on ICollection<string>, for example, will be resolved using a List<string> object. Rather than require separate services for each type, the unbound service allows mappings for all generic types to be created.

The two endpoints in Listing 14-42 request ICollection<string> and ICollection<int> services, each of which will be resolved with a different List<T> object. To target the endpoints, restart ASP.NET Core and request http://localhost:5000/string and http://localhost:5000/int. The service has been defined as a singleton, which means that the same List<string> and List<int> objects will be used to resolve all requests for ICollection<string> and ICollection<int>. Each request adds a new item to the collection, which you can see by reloading the web browser, as shown in Figure 14-16.

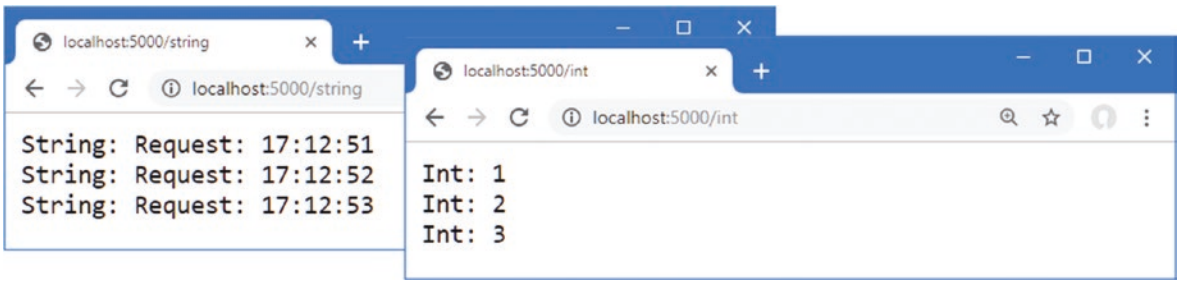

Figure 14-16. *Using a singleton service with an unbound type*

Summary

In this chapter, I described dependency injection, which is used to define services that are easy to use, easy to change, and easy to consume. I showed you the different ways that services are consumed, explained the different lifecycles that services can be given, and explained some of the less frequently used features such as dependency chains and unbound service types. In the next chapter, I describe the built-in features provided by the ASP.NET Core platform.