CENG 423 Web Application Development

Week 7 – Dependency Injection

Dependency Injection

- Services are objects that are shared between middleware components and endpoints
- They are usually used for tasks that are needed in multiple parts of the application, such as logging or database access
- *Dependency Injection* is used to create and consume services
- We are going to discuss
  - Problems that dependency injection solves
  - How dependency injection is supported by the ASP.NET Core

Dependency Injection

- 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
- How is it used
  - The Program.cs file 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

Dependency Injection

- IResponseFormatter interface
  - Defines a single method
  - Receives an HttpContext object and a string
  - To implement the interface, we create a class called TextResponseFormatter

```
namespace Platform.Services {
    public interface IResponseFormatter {

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

- TextResponseFormatter
  - 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

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

Dependency Injection

- The WeatherMiddleware file
- A typical middleware component

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

Dependency Injection

- The WeatherEndpoint file
- A typical endpoint

```
namespace Platform {
    public class WeatherEndpoint {

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

Dependency Injection

- Configuring the Program.cs file to call new files
- It calls the class-based middleware
- It routes GET requests that match the URL middleware/function, endpoint/class, and endpoint/function to the endpoints given below

```
using Platform;
using Platform.Services;

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.UseMiddleware<WeatherMiddleware>();

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

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

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

app.Run();
```

Understanding Service Location and Tight Coupling

- The service location problem
- If there is more than one component wanting to use the same service, we need to make a way to make a single service object available
- It must be easily found and consumed at every point where responses are generated
- 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, as we did in our example

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

Understanding Service Location

- The second approach is to add a static property to the service class that provides direct access to the shared instance, singleton pattern

```
private static TextResponseFormatter? shared;

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

- Configuring the Program.cs file

```
//IResponseFormatter formatter = new TextResponseFormatter();
app.MapGet("middleware/function", async (context) => {
    await TextResponseFormatter.Singleton.Format(context,
        "Middleware Function: It is snowing in Chicago");
});

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

Understanding Service Location

- The singleton pattern allows a single TextResponseFormatter to be shared
- It is used by a middleware component and an endpoint
- The single counter we have is incremented by requests from two different URLs

Understanding Service Location

- The singleton pattern is simple to understand and easy to use, but
  - the knowledge of how services are located is spread throughout the application
  - 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 Tightly Coupled Components Problem

- Our singleton pattern means that consumers are always aware of the implementation class they are using
- The class's static property is used to get the shared object
- This is not flexible
  - a change in the implementation of the IResponseFormatter interface requires locating every use of the service and replace the existing implementation class with the new one

Understanding Tightly Coupled Components Problem

- There are patterns to solve this problem, *type broker*
- A class provides access to singleton objects through their interfaces

```
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
- This pattern means that service consumers can work through interfaces rather than concrete classes

Understanding Tightly Coupled Components Problem

- 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

```
using Platform;
using Platform.Services;

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.UseMiddleware<WeatherMiddleware>();

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

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

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

app.Run();
```

Understanding Tightly Coupled Components Problem

- Service classes can focus on the features they provide without having to deal with how those features will be located

```
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

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 patterns, and is integrated with other ASP.NET Core features

```
using Platform;
using Platform.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSingleton<IResponseFormatter, HtmlResponseFormatter>();

var app = builder.Build();

app.UseMiddleware<WeatherMiddleware>();

app.MapGet("middleware/function", async (HttpContext context,
        IResponseFormatter formatter) => {
    await formatter.Format(context,"Middleware Function: It is snowing in Chicago");
});

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

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

app.Run();
```

Using Dependency Injection

- Services are registered using extension methods defined by the IServiceCollection interface
  - WebApplicationBuilder.Services
- To create a service for the IResponseFormatter interface, the extension method builder.Services.AddSingleton<IResponseFormatter, HtmlResponseFormatter>(); is used
- AddSingleton method is one of yhe extension methods available for services and tells ASP.NET Core that a single object should be used to satisfy all demands for the service
- The interface and the implementation class are specified as generic type arguments
- To consume the service async (HttpContext context, IResponseFormatter formatter) is used

Using Dependency Injection

- Many of the methods that are used to register middleware or create endpoints will accept any function which allows parameters to be defined for the services that are required to produce a response
- One consequence of this feature is that the C# compiler can't determine the parameter types, therefore, the type must be specified in the code
- The new parameter declares a dependency on the IResponseFormatter interface, and the function is said to depend on the interface
- Before the function is invoked to handle a request, 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 call to the AddSingleton method told the dependency injection system that a dependency on the IResponseFormatter interface can be resolved with an HtmlResponseFormatter object

Using Dependency Injection

- The object is created and used as an argument to invoke the handler function
- Because the object that resolves the dependency is provided from outside the 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 and consuming in the same code file may not seem impressive, but once a service is defined, it can be used almost anywhere in an ASP.NET Core application

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

Using a Service in a Middleware Class

- When the request pipeline is being set up, the ASP.NET Core platform reaches the statement in the Program.cs file that adds the WeatherMiddleware class as a component
- 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
- Two important points to understand about this example:
  - WeatherMiddleware doesn't know which implementation class will be used to resolve its dependency on the IResponseFormatter interface
  - The WeatherMiddleware class doesn't know how the dependency is resolved
    - It declares a constructor parameter and relies on ASP.NET Core to figure out the details

Using a Service in an Endpoint

- The endpoints we defined are static and does not have a constructor through which dependencies can be declared
- There are several approaches available to resolve dependencies for an endpoint class
  - Getting Services from the HttpContext Object
  - Using an Adapter Function
  - Using the Activation Utility Class

Getting Services from the HttpContext Object

```
using Platform.Services;

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");
        }
    }
}
```

Getting Services from the HttpContext Object

- The HttpContext.RequestServices property returns an object that implements the IServiceProvider interfaces, which provides access to the services that have been configured in the Program.cs file

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
- The drawback of using the HttpContext.RequestServices method is that the service must be resolved for every request that is routed to the endpoints

Using an Adapter Function

- A more elegant approach is to get the service when the endpoint's route is created and not for each request

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

```
using Platform.Services;

namespace Microsoft.AspNetCore.Builder {

    public static class EndpointExtensions {
        public static void MapWeather(this IEndpointRouteBuilder app, string path) {
            IResponseFormatter formatter =
                app.ServiceProvider.GetRequiredService<IResponseFormatter>();
            app.MapGet(path, context => Platform.WeatherEndpoint
                .Endpoint(context, formatter));
        }
    }
}
```

Using an Adapter Function

- The new file creates an extension method for the IEndpointRouterBuilder interface, which is used to create routes in the Program.cs file
- 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

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

- The MapWeather extension method sets up the route and creates the adapter around the endpoint class

Using the Activation Utility Class

- Our endpoint classes are static 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

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

Using the Activation Utility Class

- 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

```
//using Platform.Services;
using System.Reflection;

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));
        }
    }
}
```

Using the Activation Utility Class

- 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
- 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

Using the Activation Utility Class

- We need to create a route using the extension method in the Program.cs file

```
using Platform;
using Platform.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSingleton<IResponseFormatter, HtmlResponseFormatter>();

var app = builder.Build();

app.UseMiddleware<WeatherMiddleware>();

app.MapGet("middleware/function", async (HttpContext context,
        IResponseFormatter formatter) => {
    await formatter.Format(context, "Middleware Function: It is snowing in Chicago");
});

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

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

app.Run();
```