Routing, Attribute Routing, and Dependency Injection

Implementing Dependency Injection in ASP.NET Core

Objective: By the end of this activity, you will be able to implement dependency injection in an ASP.NET Core application by creating services, defining their lifetimes, and injecting them into application components like middleware and controllers.

Step 1: Prepare for the Application

You'll create a new ASP.NET Core web API project in Visual Studio Code. This application will demonstrate dependency injection by logging actions taken by different services.

Steps:

1. Open Visual Studio Code and create a new ASP.NET Core Web API project.
2. Remove any code from the existing Program.cs file.
3. Set up a basic structure for your application, including any necessary configuration.

Step 2: Create a Custom Service Interface and Implementation

Define a service interface and a class to implement this interface. This service will log messages along with a unique identifier.

Steps:

1. In Program.cs, define a new interface IMyService with a method signature void LogCreation(string message);.
2. Below the interface, create a class MyService that implements IMyService.
3. In MyService, create a private field _serviceId and initialize it in the constructor with a random six-digit number.
4. Implement the LogCreation method to print the provided message along with the service ID.

Step 3: Register the Service as a Singleton

Register your service as a singleton to make it available across the application, ensuring a single instance is used for the entire application's lifetime.

Steps:

1. In Program.cs, use builder.Services.AddSingleton<IMyService, MyService>(); to register the MyService class with IMyService.
2. Save your changes in Program.cs.

Step 4: Inject and Use the Service in a Route Handler

Inject IMyService into a route handler to log a message whenever the route is accessed.

Steps:

1. Add an endpoint in Program.cs using app.MapGet("/").
2. Define a parameter of type IMyService in this route handler and call LogCreation("Root").
3. Test your application by running it and accessing the root route to verify that the service ID logs correctly.

Step 5: Experiment with Scoped and Transient Lifetimes

Modify the service's registration type to observe differences in instance behavior for each lifetime.

Steps:

1. Replace AddSingleton in Program.cs with AddScoped and restart the application. Observe how the service instance is reused within the same request but changes across requests.
2. Next, replace AddScoped with AddTransient and observe how a new instance is created each time the service is accessed.

Step 6: Extend with Middleware to Track Service Lifecycle

Add middleware that also logs messages using IMyService, providing insight into how the service behaves in different lifecycle configurations.

Steps:

1. Add two middleware components before the endpoint in Program.cs, each using context.RequestServices.GetRequiredService<IMyService>() to get an instance of IMyService and log messages.

2. Run the application and check the console output to understand how the service instance behaves across middleware and endpoint calls under different lifetimes.

Program.cs:

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddSingleton<IMyServiceSingleton,
MyServiceSingleton>();
builder.Services.AddScoped<IMyServiceScoped, MyServiceScoped>();
builder.Services.AddTransient<IMyServiceTransient,
MyServiceTransient>();

var app = builder.Build();

app.Use(async (context, next) =>
{

context.RequestServices.GetRequiredService<IMyServiceSingleton>().LogCre
ation("Middleware 1");

context.RequestServices.GetRequiredService<IMyServiceScoped>().LogCreati
on("Middleware 1");

context.RequestServices.GetRequiredService<IMyServiceTransient>().LogCre
ation("Middleware 1");
    await next.Invoke();
});

app.Use(async (context, next) =>
{

context.RequestServices.GetRequiredService<IMyServiceSingleton>().LogCre
ation("Middleware 2");

context.RequestServices.GetRequiredService<IMyServiceScoped>().LogCreati
on("Middleware 2");

context.RequestServices.GetRequiredService<IMyServiceTransient>().LogCre
ation("Middleware 2");
    await next.Invoke();
});

app.MapGet("/", (
    IMyServiceSingleton singleton,
    IMyServiceScoped scoped,
    IMyServiceTransient transient) =>
{
    var output = new List<string>();

    output.Add(singleton.GetInfo("Root endpoint"));
    output.Add(scoped.GetInfo("Root endpoint"));
    output.Add(transient.GetInfo("Root endpoint"));

    return string.Join("\n", output);
});
```

```
app.Run();

public interface IMyServiceSingleton
{
    void LogCreation(string source);
    string GetInfo(string source);
}
public interface IMyServiceScoped
{
    void LogCreation(string source);
    string GetInfo(string source);
}
public interface IMyServiceTransient
{
    void LogCreation(string source);
    string GetInfo(string source);
}

public class MyServiceSingleton : IMyServiceSingleton
{
    private readonly int _serviceId = new Random().Next(100000, 999999);
    public void LogCreation(string source)
    {
        Console.WriteLine(GetInfo(source));
    }
    public string GetInfo(string source)
    {
        return $"[Singleton | ID: {_serviceId}] → {source} |
Explanation: Same instance across the whole application lifetime";
    }
}

public class MyServiceScoped : IMyServiceScoped
{
    private readonly int _serviceId = new Random().Next(100000, 999999);
    public void LogCreation(string source)
    {
        Console.WriteLine(GetInfo(source));
    }
    public string GetInfo(string source)
    {
        return $"[Scoped | ID: {_serviceId}] → {source} | Explanation:
Same instance reused within one HTTP request";
    }
}

public class MyServiceTransient : IMyServiceTransient
{
    private readonly int _serviceId = new Random().Next(100000, 999999);
    public void LogCreation(string source)
    {
        Console.WriteLine(GetInfo(source));
    }
    public string GetInfo(string source)
    {
        return $"[Transient | ID: {_serviceId}] → {source} |
Explanation: Always a new instance, even within the same request";
    }
}
```