



# Распределенные информационно-аналитические СИСТЕМЫ

Практическое занятие № 8. «Dependency Injection. Часть 2»

Профессор кафедры КБ-2: д.т.н. Шатовкин Р.Р.

# Учебные вопросы:

1. Применение сервисов в классах `middleware`.
2. `Scoped`-сервисы в `singleton`-объектах.
3. Множественная регистрация сервисов.

# 1. Применение сервисов в классах middleware

После добавления сервисов в коллекцию **Services** объекта **WebApplicationBuilder** они становятся доступны приложению, в том числе и в кастомных компонентах **middleware**. В **middleware** мы можем получить зависимости **тремя способами**:

Через конструктор.

Через параметр метода `Invoke/InvokeAsync`.

Через свойство `HttpContext.RequestServices`.

При этом надо учитывать, что компоненты **middleware** создаются при запуске приложения и живут в течение всего жизненного цикла приложения. То есть при последующих запросах инфраструктура ASP.NET Core использует ранее созданный компонент. И это налагает ограничения на использование зависимостей в **middleware**.

В частности, если конструктор передается **transient**-сервис, который создается при каждом обращении к нему, то при последующих запросах мы будем использовать тот же самый сервис, так как конструктор **middleware** вызывается один раз – при создании приложения.

Например, определим сервис **TimeService**:

```
1 public class TimeService
2 {
3     public TimeService()
4     {
5         Time = DateTime.Now.ToLongTimeString();
6     }
7     public string Time { get; }
8 }
```

В конструкторе устанавливается свойство, которое хранит текущее время в виде строки.

Добавим новый компонент **TimerMiddleware**, который будет использовать этот сервис для вывода времени на веб-страницу:

```

1 public class TimerMiddleware
2 {
3     RequestDelegate next;
4     TimeService timeService;
5
6     public TimerMiddleware(RequestDelegate next, TimeService timeService)
7     {
8         this.next = next;
9         this.timeService = timeService;
10    }
11
12    public async Task InvokeAsync(HttpContext context)
13    {
14        if (context.Request.Path == "/time")
15        {
16            context.Response.ContentType = "text/html; charset=utf-8";
17            await context.Response.WriteAsync($"Текущее время: {timeService?.Time}");
18        }
19        else
20        {
21            await next.Invoke(context);
22        }
23    }
24 }

```

Если сервис **TimeService** добавляется в коллекцию сервисов приложения, то мы сможем получить его через конструктор класса **TimerMiddleware**.

Логика компонента предполагает, что, если запрос пришел по адресу **"/time"**, то с помощью **TimeService** возвращается текущее время. Иначе мы просто обращаемся к следующему **middleware** в конвейере обработки запроса.

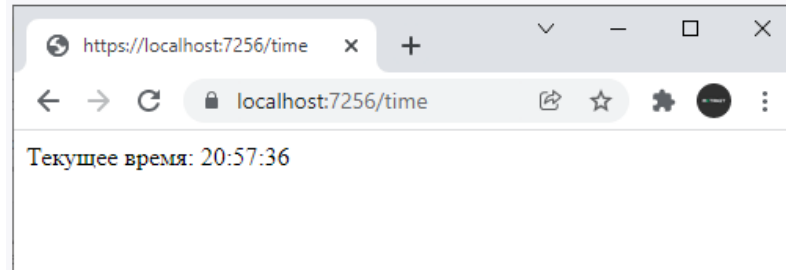
Используем этот компонент в файле **Program.cs**:

```

1 var builder = WebApplication.CreateBuilder();
2
3 builder.Services.AddTransient<TimeService>();
4
5 var app = builder.Build();
6
7 app.UseMiddleware<TimerMiddleware>();
8 app.Run(async (context) => await context.Response.WriteAsync("Hello METANIT.COM"));
9
10 app.Run();

```

В итоге, если мы обратимся по пути **"/time"**, то приложение выведет текущее время:



Однако сколько бы мы раз не обращались по этому пути, мы все время будем получать одно и то же время, так как объект **TimerMiddleware** был создан еще при первом запросе. Поэтому передача через конструктор **middleware** больше подходит для сервисов с жизненным циклом **Singleton**, которые создаются один раз для всех последующих запросов.

Если же в **middleware** необходимо использовать сервисы с жизненным циклом **Scoped** или **Transient**, то лучше их передавать через параметр метода **Invoke/InvokeAsync**:

```
1 public class TimerMiddleware
2 {
3     RequestDelegate next;
4
5     public TimerMiddleware(RequestDelegate next)
6     {
7         this.next = next;
8     }
9
10    public async Task InvokeAsync(HttpContext context, TimeService timeService)
11    {
12        if (context.Request.Path == "/time")
13        {
14            context.Response.ContentType = "text/html; charset=utf-8";
15            await context.Response.WriteAsync($"Текущее время: {timeService?.Time}");
16        }
17        else
18        {
19            await next.Invoke(context);
20        }
21    }
22 }
```

## 2. Scoped-сервисы в singleton-объектах

Все объекты, которые используются в ASP.NET Core, имеют три варианта жизненного цикла. **Singleton**-объекты создаются один раз при запуске приложения, и при всех запросах к приложению оно использует один и тот же **singleton**-объект. К подобным **singleton**-объектам относятся, к примеру, компоненты **middleware** или сервисы, которые регистрируются с помощью метода **AddSingleton()**.

**Transient**-объекты создаются каждый раз, когда нам требуется экземпляр определенного класса. А **scoped**-объекты создаются по одному на каждый запрос.

Одни объекты или сервисы с помощью встроенного механизма **dependency injection** можно передать в другие объекты. Наиболее распространенный способ внедрения объектов представляет инъекция через конструктор. Однако начиная с версии ASP.NET Core 2.0 мы не можем передавать **scoped**-сервисы в конструктор **singleton**-объектов.

Например, пусть будут определены следующие классы:

```
1 public interface ITimer
2 {
3     string Time { get; }
4 }
5 public class Timer : ITimer
6 {
7     public Timer()
8     {
9         Time = DateTime.Now.ToString();
10    }
11    public string Time { get; }
12 }
13 public class TimeService
14 {
15     private ITimer timer;
16     public TimeService(ITimer timer)
17     {
18         this.timer = timer;
19     }
20     public string GetTime() => timer.Time;
21 }
```

**TimeService** получает через конструктор сервис **ITimer** и использует его для получения текущего времени.

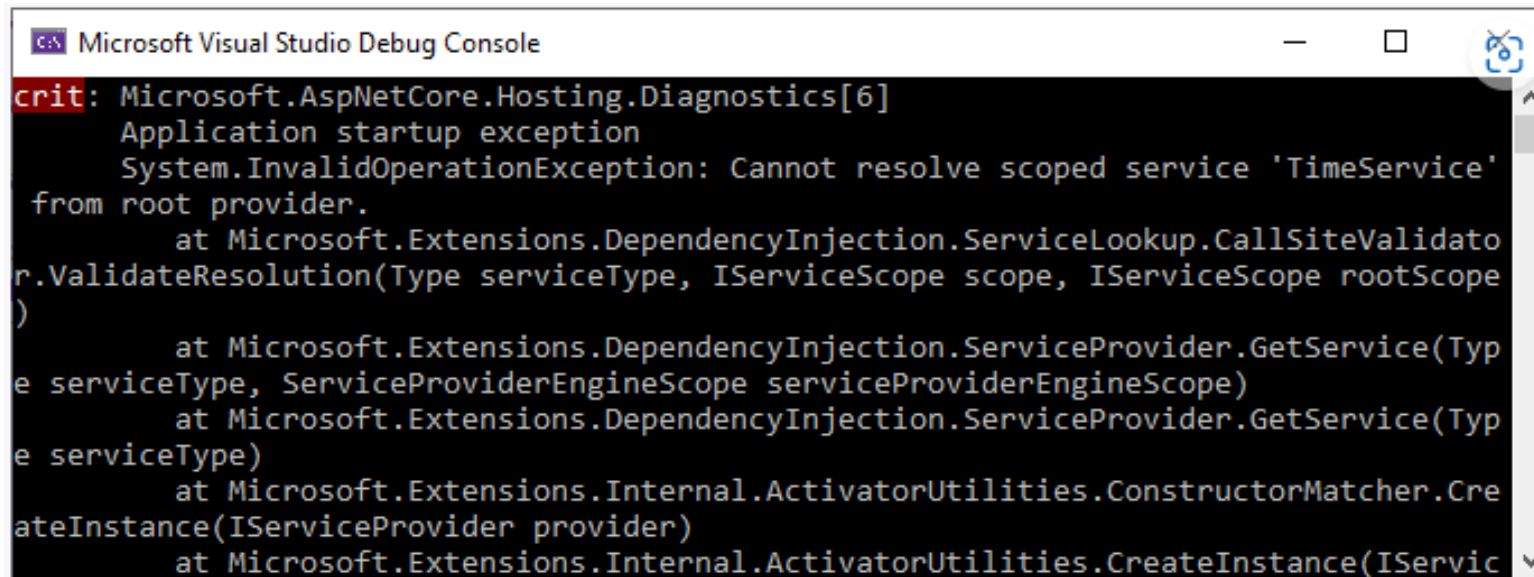
Также пусть будет определен компонент **middleware TimerMiddleware**:

```
1 public class TimerMiddleware
2 {
3     TimeService timeService;
4     public TimerMiddleware(RequestDelegate next, TimeService timeService)
5     {
6         this.timeService = timeService;
7     }
8
9     public async Task Invoke(HttpContext context)
10    {
11        await context.Response.WriteAsync($"Time: {timeService?.GetTime()}");
12    }
13 }
```

Компонент **TimerMiddleware** получает сервис **TimeService** и отправляет в ответ клиенту информацию о текущем времени. **TimerMiddleware** является **singleton**-объектом. И теперь регистрируем сервис **TimeService** как **scoped**-объект:

```
1 var builder = WebApplication.CreateBuilder();
2
3 builder.Services.AddTransient<ITimer, Timer>();
4 builder.Services.AddScoped<TimeService>();
5
6 var app = builder.Build();
7
8 app.UseMiddleware<TimerMiddleware>();
9
10 app.Run();
```

Если мы запустим приложение, то консоль приложения нам отобразит ошибку типа **"InvalidOperationException: Cannot resolve scoped service 'TimeService' from root provider."**:



```
crit: Microsoft.AspNetCore.Hosting.Diagnostics[6]
Application startup exception
System.InvalidOperationException: Cannot resolve scoped service 'TimeService'
from root provider.
    at Microsoft.Extensions.DependencyInjection.ServiceLookup.CallSiteValidator.ValidateResolution(Type serviceType, IServiceScope scope, IServiceScope rootScope)
    at Microsoft.Extensions.DependencyInjection.ServiceProvider.GetService(Type serviceType, ServiceProviderEngineScope serviceProviderEngineScope)
    at Microsoft.Extensions.DependencyInjection.ServiceProvider.GetService(Type serviceType)
    at Microsoft.Extensions.Internal.ActivatorUtilities.ConstructorMatcher.CreateInstance(IServiceProvider provider)
    at Microsoft.Extensions.Internal.ActivatorUtilities.CreateInstance(IServiceProvider provider, Type instanceType, Object[] parameters)
```

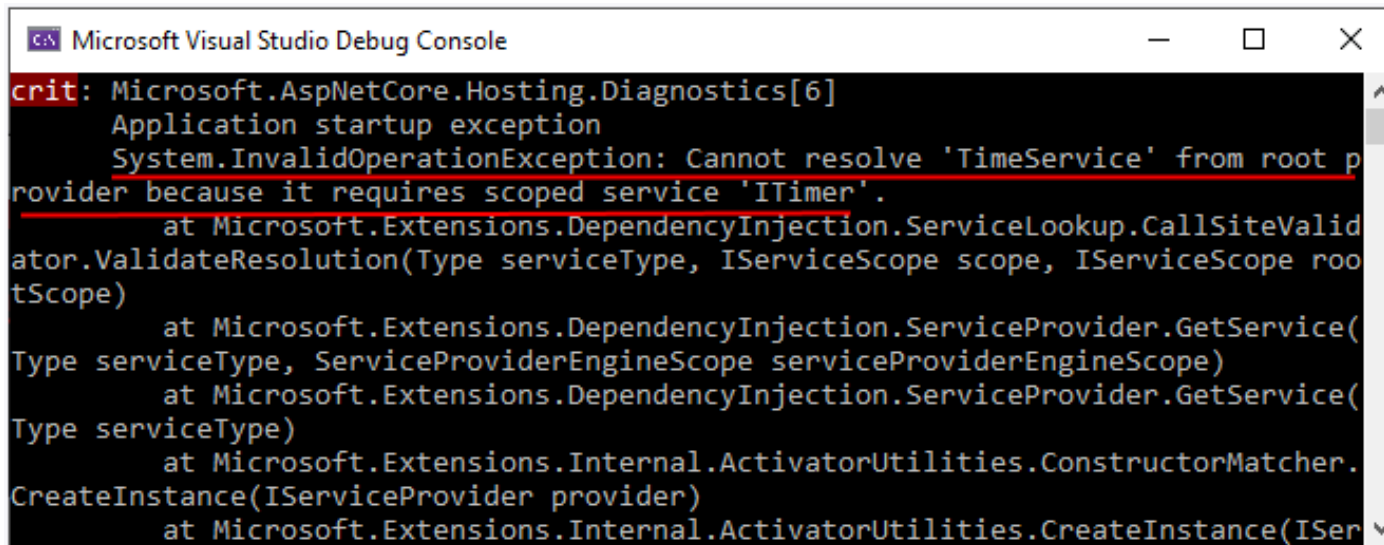
То есть на момент создания объекта **TimerMiddleware** **scoped**-сервис **TimeService** еще не установлен, соответственно он использоваться не может. А без создания объекта **TimeService** нельзя создать объект **TimerMiddleware**.

Аналогичная ситуация может возникнуть, если **TimeService** добавляется как **Transient**, а сервис **ITimer** определен как **Scoped**:

```
1 var builder = WebApplication.CreateBuilder();
2
3 builder.Services.AddScoped<ITimer, Timer>();
4 builder.Services.AddTransient<TimeService>();
5
6 var app = builder.Build();
7
8 app.UseMiddleware<TimerMiddleware>();
9
10 app.Run();
```

В этом случае для создания объекта **TimeService** надо получить сервис **ITimer**, но на момент вызова конструктора **TimerMiddleware** сервис **ITimer** еще не определен:





```
Microsoft Visual Studio Debug Console

crit: Microsoft.AspNetCore.Hosting.Diagnostics[6]
      Application startup exception
      System.InvalidOperationException: Cannot resolve 'TimeService' from root p
      rovider because it requires scoped service 'ITimer'.
         at Microsoft.Extensions.DependencyInjection.ServiceLookup.CallSiteValid
         ator.ValidateResolution(Type serviceType, IServiceScope scope, IServiceScope rootScope)
         at Microsoft.Extensions.DependencyInjection.ServiceProvider.GetService(
         Type serviceType, ServiceProviderEngineScope serviceProviderEngineScope)
         at Microsoft.Extensions.DependencyInjection.ServiceProvider.GetService(
         Type serviceType)
         at Microsoft.Extensions.Internal.ActivatorUtilities.ConstructorMatcher.
         CreateInstance(IServiceProvider provider)
         at Microsoft.Extensions.Internal.ActivatorUtilities.CreateInstance(ISer
```

Для выхода из этой ситуации ни **TimeService**, ни **ITimer** не должны иметь жизненный цикл **Scoped**. То есть это может быть **Transient** или **Singleton**.

Рассмотрим еще одну ситуацию, с которой можно столкнуться в любой части приложения, а не только в конструкторе **middleware**, когда сервис **TimeService** представляет **singleton**, а **ITimer** – **scoped**-объект:

```
1 var builder = WebApplication.CreateBuilder();
2
3 builder.Services.AddScoped<ITimer, Timer>();
4 builder.Services.AddSingleton<TimeService>();
5
6 var app = builder.Build();
7
8 app.UseMiddleware<TimerMiddleware>();
9
10 app.Run();
```

И, допустим, эти сервисы используются в **TimerMiddleware** непосредственно при обработке запроса в методе **Invoke/InvokeAsync**:

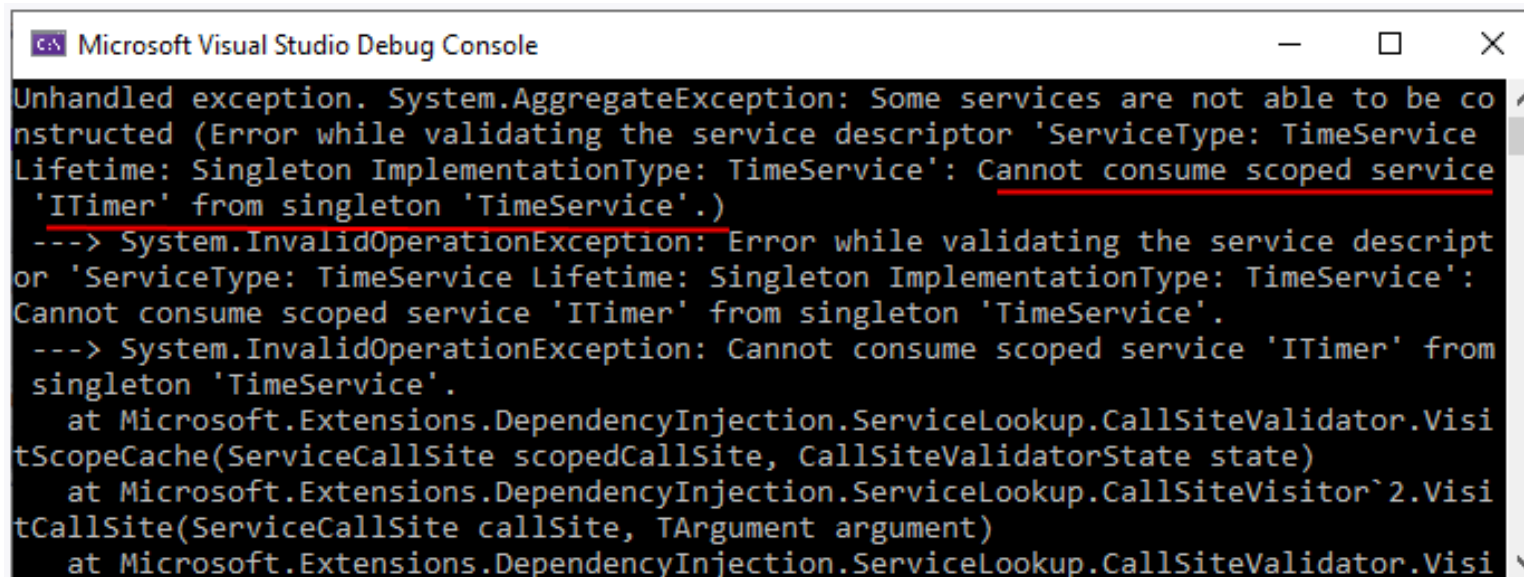
```

1 public class TimerMiddleware
2 {
3     public TimerMiddleware(RequestDelegate next) { }
4
5     public async Task Invoke(HttpContext context, TimeService timeService)
6     {
7         await context.Response.WriteAsync($"Time: {timeService?.GetTime()}");
8     }
9 }

```

При запуске приложения мы опять же столкнемся с ошибкой, только немного другой "Cannot consume scoped service 'DIApp.ITimer' from singleton 'DIApp.TimeService'".

Но суть будет та же самая – мы не можем по умолчанию передавать в конструктор **singleton**-объекта **scoped**-сервис.



```

Microsoft Visual Studio Debug Console
Unhandled exception. System.AggregateException: Some services are not able to be co
nstructed (Error while validating the service descriptor 'ServiceType: TimeService
Lifetime: Singleton ImplementationType: TimeService': Cannot consume scoped service
'ITimer' from singleton 'TimeService'.)
---> System.InvalidOperationException: Error while validating the service descript
or 'ServiceType: TimeService Lifetime: Singleton ImplementationType: TimeService':
Cannot consume scoped service 'ITimer' from singleton 'TimeService'.
---> System.InvalidOperationException: Cannot consume scoped service 'ITimer' from
singleton 'TimeService'.
    at Microsoft.Extensions.DependencyInjection.ServiceLookup.CallSiteValidator.Visi
tScopeCache(ServiceCallSite scopedCallSite, CallSiteValidatorState state)
    at Microsoft.Extensions.DependencyInjection.ServiceLookup.CallSiteVisitor`2.Visi
tCallSite(ServiceCallSite callSite, TArgument argument)
    at Microsoft.Extensions.DependencyInjection.ServiceLookup.CallSiteValidator.Visi

```

### 3. Множественная регистрация сервисов

По умолчанию при внедрении зависимостей в ASP.NET Core одна зависимость сопоставляется с одним типом. Однако бывают ситуации, когда требуется отойти от этой привязки один к одному.

Первая ситуация: для одной зависимости необходимо зарегистрировать сразу несколько конкретных реализаций.

Вторая ситуация: для нескольких зависимостей необходимо зарегистрировать один и тот же объект.

#### Регистрация для одной зависимости нескольких типов

ASP.NET Core позволяет зарегистрировать для одной зависимости сразу несколько типов. Рассмотрим простейший пример:

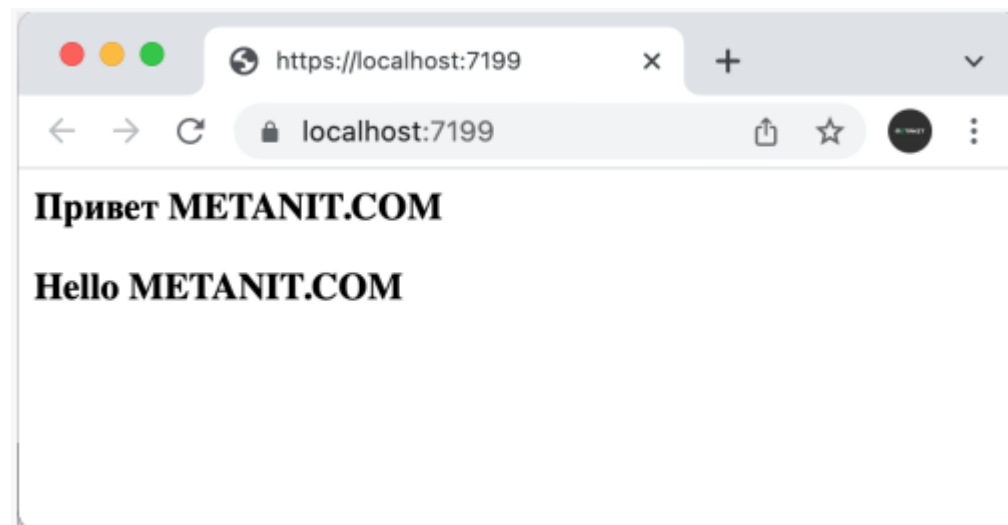
```
1 var builder = WebApplication.CreateBuilder();
2 builder.Services.AddTransient<IHelloService, RuHelloService>();
3 builder.Services.AddTransient<IHelloService, EnHelloService>();
4
5 var app = builder.Build();
6
7 app.UseMiddleware<HelloMiddleware>();
8
9 app.Run();
10
11
12 interface IHelloService
13 {
14     string Message { get; }
15 }
16
17 class RuHelloService : IHelloService
18 {
19     public string Message => "Привет METANIT.COM";
20 }
21 class EnHelloService : IHelloService
22 {
23     public string Message => "Hello METANIT.COM";
24 }
25
26 class HelloMiddleware
27 {
28     readonly IEnumerable<IHelloService> helloServices;
29
30     public HelloMiddleware(RequestDelegate _, IEnumerable<IHelloService> helloServices)
31     {
32         this.helloServices = helloServices;
33     }
34
35     public async Task InvokeAsync(HttpContext context)
36     {
37         context.Response.ContentType = "text/html; charset=utf-8";
38         string responseText = "";
39         foreach (var service in helloServices)
40         {
41             responseText += $"<h3>{service.Message}</h3>";
42         }
43         await context.Response.WriteAsync(responseText);
44     }
45 }
```

Здесь интерфейс **IHelloService** с помощью свойства **Message** определяет некоторое сообщение. Этот интерфейс реализуется двумя классами: **RuHelloService** и **EnHelloService**, каждый из которых определяет свое сообщение. И оба этих класса регистрируются в коллекции сервисов в качестве реализаций для сервиса **IHelloService**. Далее все эти реализации можно получить в виде коллекции **IEnumerable<IHelloService>**.

Этот сервис применяется в **middleware HelloMiddleware**, который внедряется в конвейер обработки запроса и обрабатывает все запросы к приложению. И в его конструкторе получаем все зарегистрированные реализации сервиса **IHelloService** через объект **IEnumerable<IHelloService>**.

```
1 public HelloMiddleware(RequestDelegate _, IEnumerable<IHelloService> helloServices)
2 {
3     this.helloServices = helloServices;
4 }
```

Соответственно далее все зарегистрированные реализации сервиса **IHelloService** можно использовать для обработки запроса:



## Регистрация одного объекта для нескольких зависимостей

Теперь рассмотрим другую ситуацию: использование несколькими зависимостями одного и того же объекта. Сначала рассмотрим ситуацию, с которой мы можем столкнуться. Допустим, у нас есть следующие объекты:

```
1 interface IGenerator
2 {
3     int GenerateValue();
4 }
5 interface IReader
6 {
7     int ReadValue();
8 }
9 class ValueStorage : IGenerator, IReader
10 {
11     int value;
12     public int GenerateValue()
13     {
14         value = new Random().Next();
15         return value;
16     }
17
18     public int ReadValue() => value;
19 }
```

Здесь интерфейс **IGenerator** предназначен для генерации некоторого числа, а интерфейс **IReader** – для чтения некоторого числа. Класс **ValueStorage** реализует оба этих интерфейса: в методе **Generate** изменяет значение переменной **value** и возвращает его, а в методе **Read** просто возвращает значение переменной **value**. То есть метод **Read** считывает текущее значение **value**, а метод **Generate** изменяет его.

Используем эти сервисы в приложении:

```
1 var builder = WebApplication.CreateBuilder();
2 builder.Services.AddSingleton<IGenerator, ValueStorage>();
3 builder.Services.AddSingleton<IReader, ValueStorage>();
4
5 var app = builder.Build();
6
7 app.UseMiddleware<GeneratorMiddleware>();
8 app.UseMiddleware<ReaderMiddleware>();
9
10 app.Run();
11
12 class GeneratorMiddleware
13 {
14     RequestDelegate next;
15     IGenerator generator;
16
17     public GeneratorMiddleware(RequestDelegate next, IGenerator generator)
18     {
19         this.next = next;
20         this.generator = generator;
21     }
22     public async Task InvokeAsync(HttpContext context)
23     {
24         if (context.Request.Path == "/generate")
25             await context.Response.WriteAsync($"New Value: {generator.GenerateValue()}");
26         else
27             await next.Invoke(context);
28     }
29 }
```

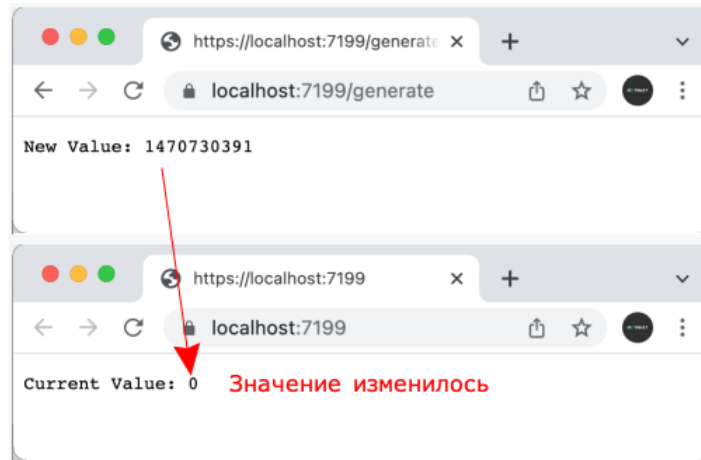
```

30 class ReaderMiddleware
31 {
32     IReader reader;
33
34     public ReaderMiddleware(RequestDelegate _, IReader reader) => this.reader = reader;
35
36     public async Task InvokeAsync(HttpContext context)
37     {
38         await context.Response.WriteAsync($"Current Value: {reader.ReadValue()}");
39     }
40 }
41
42 interface IGenerator
43 {
44     int GenerateValue();
45 }
46 interface IReader
47 {
48     int ReadValue();
49 }
50 class ValueStorage : IGenerator, IReader
51 {
52     int value;
53     public int GenerateValue()
54     {
55         value = new Random().Next();
56         return value;
57     }
58
59     public int ReadValue() => value;
60 }

```

Здесь для обеих зависимостей – **IGenerator** и **IReader** определена одна реализация – **ValueStorage**. При обращении по адресу **"/generate"** срабатывает **middleware GenerateMiddleware**, который получает сервис **IGenerator** и с его помощью генерирует новое значение.

При обращении по всем иным адресам срабатывает **middleware ReaderMiddleware**, который получает сервис **IReader** и возвращает текущее значение. Однако при запуске проекта мы увидим, что генерируемое значение и возвращаемое значения никак не синхронизированы, потому что, несмотря на то, что оба сервиса представляют синглтоны, они используют два разных экземпляра класса **ValueStorage**:



Для исправления ситуации нам надо определить один объект для обеих зависимостей. Это можно сделать, например, следующим образом:

```
1 var builder = WebApplication.CreateBuilder();
2
3 builder.Services.AddSingleton<ValueStorage>();
4 builder.Services.AddSingleton<IGenerator>(serv => serv.GetRequiredService<ValueStorage>());
5 builder.Services.AddSingleton<IReader>(serv => serv.GetRequiredService<ValueStorage>());
6
7 var app = builder.Build();
8
9 app.UseMiddleware<GeneratorMiddleware>();
10 app.UseMiddleware<ReaderMiddleware>();
11
12 app.Run();
13
14 class GeneratorMiddleware
15 {
16     RequestDelegate next;
17     IGenerator generator;
18
19     public GeneratorMiddleware(RequestDelegate next, IGenerator generator)
20     {
21         this.next = next;
22         this.generator = generator;
23     }
24     public async Task InvokeAsync(HttpContext context)
25     {
26         if (context.Request.Path == "/generate")
27             await context.Response.WriteAsync($"New Value: {generator.GenerateValue()}");
28         else
29             await next.Invoke(context);
30     }
31 }
```



```

32 class ReaderMiddleware
33 {
34     IReader reader;
35
36     public ReaderMiddleware(RequestDelegate _, IReader reader) => this.reader = reader;
37
38     public async Task InvokeAsync(HttpContext context)
39     {
40         await context.Response.WriteAsync($"Current Value: {reader.ReadValue()}");
41     }
42 }
43
44 interface IGenerator
45 {
46     int GenerateValue();
47 }
48 interface IReader
49 {
50     int ReadValue();
51 }
52 class ValueStorage : IGenerator, IReader
53 {
54     int value;
55     public int GenerateValue()
56     {
57         value = new Random().Next();
58         return value;
59     }
60
61     public int ReadValue() => value;
62 }

```

Теперь определяем один объект **ValueStorage** в виде **singleton** -сервиса:

```

1 builder.Services.AddSingleton<ValueStorage>();

```

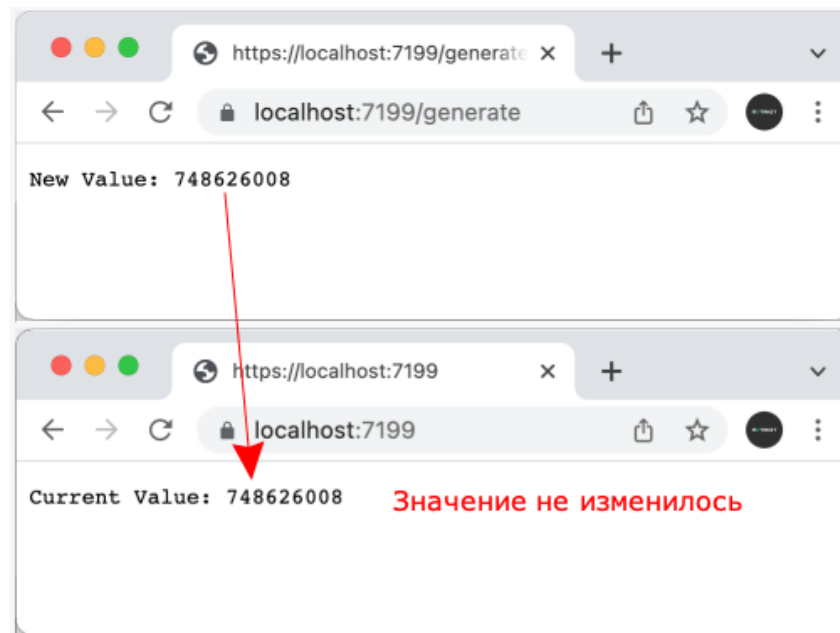
Затем получаем его из коллекции сервисов и устанавливаем в качестве реализации для обоих зависимостей:

```

1 builder.Services.AddSingleton<IGenerator>(serv => serv.GetRequiredService<ValueStorage>());
2 builder.Services.AddSingleton<IReader>(serv => serv.GetRequiredService<ValueStorage>());

```

Соответственно теперь мы получим другие результаты:



В качестве альтернативы можно было бы создать объект во вне и передать его сервисам:

```
1 var valueStorage = new ValueStorage();  
2 builder.Services.AddSingleton<IGenerator>(_ => valueStorage);  
3 builder.Services.AddSingleton<IReader>(_ => valueStorage);
```