



Распределенные информационно-аналитические СИСТЕМЫ

Практическое занятие № 7. «Dependency Injection. Часть 1»

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

Учебные вопросы:

1. Внедрение зависимостей и IServiceCollection.
2. Создание сервисов.
3. Получение зависимостей.
4. Жизненный цикл зависимостей.

1. Внедрение зависимостей и IServiceCollection

Dependency injection (DI) или внедрение зависимостей представляет механизм, который позволяет сделать взаимодействующие в приложении объекты слабосвязанными. Такие объекты связаны между собой через абстракции, например, через интерфейсы, что делает всю систему более гибкой, более адаптируемой и расширяемой.

В центре подобного механизма находится понятие зависимость – некоторая сущность, от которой зависит другая сущность. Например:

```
1 class Logger
2 {
3     public void Log(string message) => Console.WriteLine(message);
4 }
5 class Message
6 {
7     Logger logger = new Logger();
8     public string Text { get; set; } = "";
9     public void Print() => logger.Log(Text);
10 }
```

Здесь сущность **Message**, которая представляет некоторое сообщение, зависит от другой сущности – **Logger**, которая представляет логгер. В методе **Print()** класса **Message** имитируется логгирование текста сообщения путем вызова у объекта **Logger** метода **Log**, который выводит сообщение на консоль. Однако здесь класс **Message** тесно связан с классом **Logger**. Класс **Message** отвечает за создание объекта **Logger**. Это имеет ряд недостатков. Прежде всего, если мы захотим вместо класса **Logger** использовать другой тип тип логгера, например, логгировать в файл, а не на консоль, то нам придется менять класс **Message**. Один класс не составит труда поменять, но если в проекте таких классов много, то поменять во всех класс **Logger** на другой будет труднее. Кроме того, класс **Logger** может иметь свои зависимости, которые тоже может потребоваться поменять. В итоге такими системами сложнее управлять и сложнее тестировать.

Чтобы отвязать объект **Logger** от класса **Message**, мы можем создать абстракцию, которая будет представлять логгер, и передавать ее извне в объект **Message**:

```
1 interface ILogger
2 {
3     void Log(string message);
4 }
5 class Logger : ILogger
6 {
7     public void Log(string message) => Console.WriteLine(message);
8 }
9 class Message
10 {
11     ILogger logger;
12     public string Text { get; set; } = "";
13     public Message(ILogger logger)
14     {
15         this.logger = logger;
16     }
17     public void Print() => logger.Log(Text);
18 }
```

Теперь класс **Message** не зависит от конкретной реализации класса **Logger** – это может быть любая реализация интерфейса **ILogger**. Кроме того, создание объекта логгера выносится во внешний код. Класс **Message** больше ничего не знает о логгере кроме того, что у него есть метод **Log**, который позволяет логгировать его текст.

Тем не менее остается проблема управления подобными зависимостями, особенно если это касается больших приложений. Нередко для установки зависимостей в подобных системах используются специальные контейнеры – **IoC**-контейнеры (**Inversion of Control**). Такие контейнеры служат своего рода фабриками, которые устанавливают зависимости между абстракциями и конкретными объектами и, как правило, управляют созданием этих объектов.

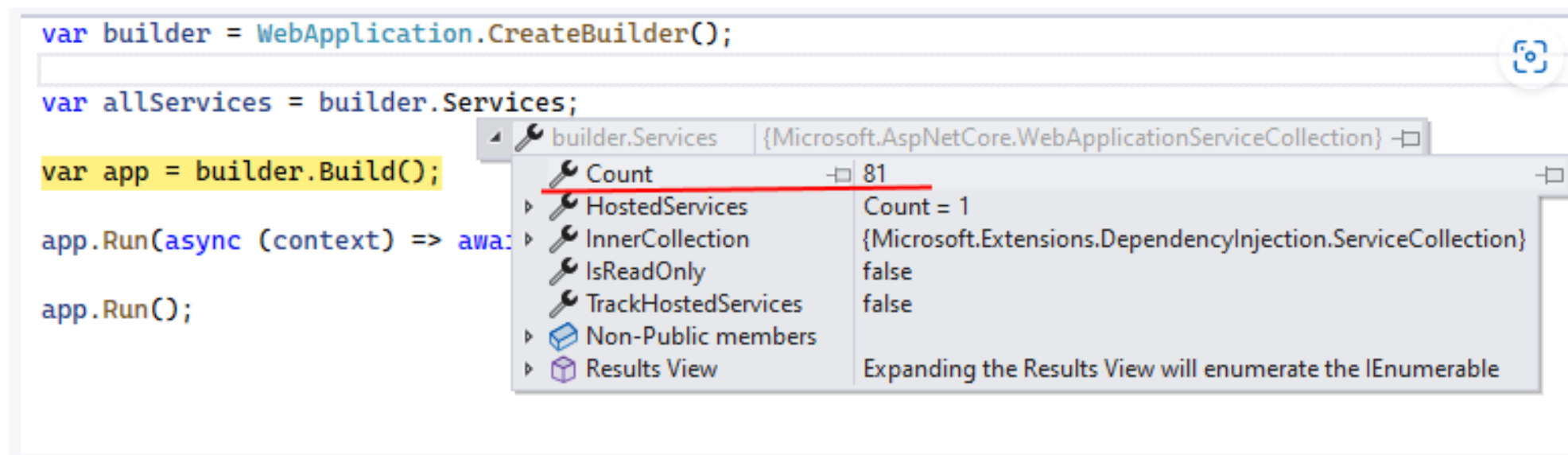
Преимуществом ASP.NET Core в этом отношении является то, что фреймворк уже по умолчанию имеет встроенный контейнер внедрения зависимостей, который представлен интерфейсом **IServiceProvider**. А сами зависимости еще называются сервисами, собственно поэтому контейнер можно назвать провайдером сервисов. Этот контейнер отвечает за сопоставление зависимостей с конкретными типами и за внедрение зависимостей в различные объекты.

Установка встроенных сервисов фреймворка

За управление сервисами в приложении в классе **WebApplicationBuilder** определено свойство **Services**, которое представляет объект **IServiceCollection** – коллекцию сервисов:

```
1 WebApplicationBuilder builder = WebApplication.CreateBuilder();
2 IServiceCollection allServices = builder.Services; // коллекция сервисов
```

И даже если мы не добавляем в эту коллекцию никаких сервисов, **IServiceCollection** уже содержит ряд сервисов по умолчанию



```
var builder = WebApplication.CreateBuilder();
var allServices = builder.Services;
var app = builder.Build();
app.Run(async (context) => await ...);
app.Run();
```

Property	Value
Count	81
HostedServices	Count = 1
InnerCollection	{Microsoft.Extensions.DependencyInjection.ServiceCollection}
IsReadOnly	false
TrackHostedServices	false
Non-Public members	
Results View	Expanding the Results View will enumerate the IEnumerable

Как видно на скриншоте, в коллекции **IServiceCollection** 81 сервис, который мы можем использовать в приложении. Это такие сервисы, как **ILogger<T>**, **ILoggerFactory**, **IWebHostEnvironment** и ряд других. Они добавляются по умолчанию инфраструктурой ASP.NET Core. И мы их можем использовать в различных частях приложения.

Информация о сервисах

Каждый сервис в коллекции **IServiceCollection** представляет объект **ServiceDescriptor**, который несет некоторую информацию.

В частности, наиболее важные **свойства этого объекта**:

ServiceType: тип сервиса.

ImplementationType: тип реализации сервиса.

Lifetime: жизненный цикл сервиса.

Например, получим все сервисы, которые добавлены в приложение:

```
1 using System.Text;
2
3 var builder = WebApplication.CreateBuilder();
4
5 var services = builder.Services;
6
7 var app = builder.Build();
8
9 app.Run(async context =>
10 {
11     var sb = new StringBuilder();
12     sb.Append("<h1>Все сервисы</h1>");
13     sb.Append("<table>");
14     sb.Append("<tr><th>Тип</th><th>Lifetime</th><th>Реализация</th></tr>");
15     foreach (var svc in services)
16     {
17         sb.Append("<tr>");
18         sb.Append($"<td>{svc.ServiceType.FullName}</td>");
19         sb.Append($"<td>{svc.Lifetime}</td>");
20         sb.Append($"<td>{svc.ImplementationType?.FullName}</td>");
21         sb.Append("</tr>");
22     }
23     sb.Append("</table>");
24     context.Response.ContentType = "text/html;charset=utf-8";
25     await context.Response.WriteAsync(sb.ToString());
26 });
27
28 app.Run();
```

Тип	Lifetime	Реализация
Microsoft.Extensions.Hosting.IHostingEnvironment	Singleton	
Microsoft.Extensions.Hosting.IHostEnvironment	Singleton	
Microsoft.Extensions.Hosting.HostBuilderContext	Singleton	
Microsoft.Extensions.Configuration.IConfiguration	Singleton	
Microsoft.Extensions.Hosting.IApplicationLifetime	Singleton	
Microsoft.Extensions.Hosting.IHostApplicationLifetime	Singleton	Microsoft.Extensions.Hosting.Internal.ApplicationLifetime
Microsoft.Extensions.Hosting.IHostLifetime	Singleton	Microsoft.Extensions.Hosting.Internal.ConsoleLifetime
Microsoft.Extensions.Hosting.IHost	Singleton	
Microsoft.Extensions.Options.IOptions`1	Singleton	Microsoft.Extensions.Options.UnnamedOptionsManager`1
Microsoft.Extensions.Options.IOptionsSnapshot`1	Scoped	Microsoft.Extensions.Options.OptionsManager`1
Microsoft.Extensions.Options.IOptionsMonitor`1	Singleton	Microsoft.Extensions.Options.OptionsMonitor`1

Регистрация встроенных сервисов ASP.NET Core

Кроме ряда подключаемых по умолчанию сервисов ASP.NET Core имеет еще ряд встроенных сервисов, которые мы можем подключать в приложение при необходимости. Все сервисы и компоненты **middleware**, которые предоставляются ASP.NET по умолчанию, регистрируются в приложение с помощью методов расширений **IServiceCollection**, имеющих общую форму **Add[название_сервиса]**.

Например:

```
1 var builder = WebApplication.CreateBuilder();
2 builder.Services.AddMvc();
```

Для объекта **IServiceCollection** определено ряд методов расширений, которые начинаются на **Add**, как, например, **AddMvc()**. Эти методы добавляют в объект **IServiceCollection** соответствующие сервисы. Например, **AddMvc()** добавляет в приложение сервисы **MVC**, благодаря чему мы сможем их использовать в приложении.

2. Создание сервисов

Фреймворк ASP.NET Core предоставляет ряд встроенных сервисов, которые мы можем использовать. Но также мы можем создавать свои собственные сервисы. Рассмотрим, как это сделать.

Определим новый интерфейс **ITimeService**, который предназначен для получения времени:

```
1 interface ITimeService
2 {
3     string GetTime();
4 }
```

И также определим два класса, которые будут реализовать данный интерфейс. Первый класс будет называться **ShortTimeService** и будет возвращать текущее время в формате **hh:mm** (то есть часы и минуты):

```
1 // время в формате hh:mm
2 class ShortTimeService : ITimeService
3 {
4     public string GetTime() => DateTime.Now.ToShortTimeString();
5 }
```

Второй класс будет называться **LongTimeService** — он будет возвращать время в формате **hh:mm:ss**:

```
1 // время в формате hh:mm:ss
2 class LongTimeService : ITimeService
3 {
4     public string GetTime() => DateTime.Now.ToLongTimeString();
5 }
```

Теперь добавим в коллекцию сервисов сервис **ITimeService** и используем его в приложении:


```

1 var builder = WebApplication.CreateBuilder();
2
3 builder.Services.AddTransient<ITimeService, ShortTimeService>();
4
5 var app = builder.Build();
6
7 app.Run(async context =>
8 {
9     var timeService = app.Services.GetService<ITimeService>();
10    await context.Response.WriteAsync($"Time: {timeService?.GetTime()}");
11 });
12
13 app.Run();
14
15 interface ITimeService
16 {
17     string GetTime();
18 }
19 // время в формате hh:mm
20 class ShortTimeService : ITimeService
21 {
22     public string GetTime() => DateTime.Now.ToShortTimeString();
23 }
24 // время в формате hh:mm:ss
25 class LongTimeService : ITimeService
26 {
27     public string GetTime() => DateTime.Now.ToLongTimeString();
28 }

```

Здесь надо выделить два момента. Во-первых, добавление сервиса в коллекцию сервисов приложения:

```

1 builder.Services.AddTransient<ITimeService, ShortTimeService>();

```

Благодаря вызову **AddTransient<ITimeService, ShortTimeService>()** система на место объектов интерфейса **ITimeService** будет передавать экземпляры класса **ShortTimeService**.

Кроме того, сервисы добавляются до создания объекта **WebApplication** методом **Build()** объекта **WebApplicationBuilder**:

```

1 // добавление сервисов
2 builder.Services.AddTransient<ITimeService, ShortTimeService>();
3 // создание объекта WebApplication
4 var app = builder.Build();

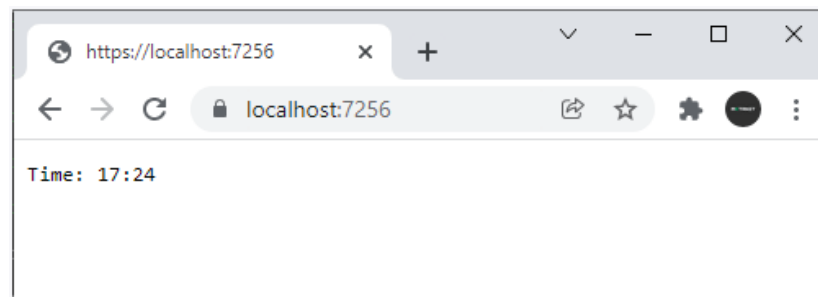
```

После добавления сервиса его можно получить и использовать в любой части приложения. Для получения сервиса могут применяться различные способы в зависимости от ситуации. В данном случае используется свойство **app.Services.**, которое предоставляет провайдер сервисов – объект **IServiceProvider**. Для получения сервиса у провайдера сервиса вызывается метод **GetService()**, который типизируется типом сервиса:

```
1 var timeService = app.Services.GetService<ITimeService>();
```

После получения сервиса мы можем использовать его.

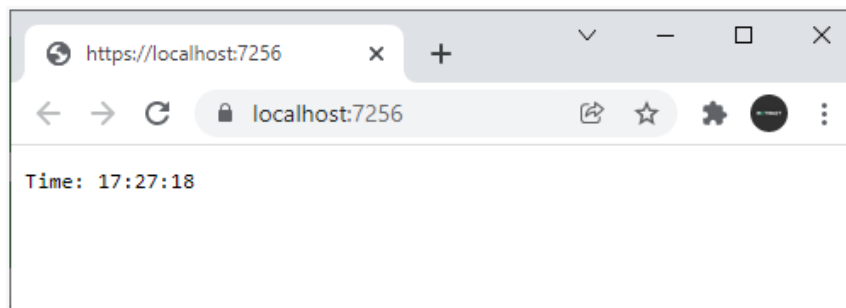
```
1 await context.Response.WriteAsync($"Time: {timeService?.GetTime()}");
```



Поскольку метод **AddTransient** установил зависимость между **ITimeService** и **ShortTimeService**, то в браузере выводится текущее время в формате **"hh:mm"**. Мы можем поменять тип, сопоставляемый с **ITimeService**:

```
1 builder.Services.AddTransient<ITimeService, LongTimeService>();
```

И в этом случае мы увидим другое сообщение.



Сервис как конкретный класс

При этом необязательно разделять определение сервиса в виде интерфейса и его реализацию. Сам термин "сервис" в данном случае может представлять любой объект, функциональность которого может использоваться в приложении.

Например, определим новый класс **TimeService**:

```
1 public class TimeService
2 {
3     public string GetTime() => DateTime.Now.ToShortTimeString();
4 }
```

Данный класс определяет один метод **GetTime()**, который возвращает текущее время. Используем этот класс в качестве сервиса:

```
1 var builder = WebApplication.CreateBuilder();
2
3 builder.Services.AddTransient<TimeService>();
4
5 var app = builder.Build();
6 app.Run(async context =>
7 {
8     var timeService = app.Services.GetService<TimeService>();
9     await context.Response.WriteAsync($"Time: {timeService?.GetTime()}");
10 });
11
12 app.Run();
13
14 public class TimeService
15 {
16     public string GetTime() => DateTime.Now.ToShortTimeString();
17 }
```

Для добавления сервиса в эту коллекцию применяется метод **AddTransient()**:

```
1 builder.Services.AddTransient<TimeService>();
```

После добавления сервиса мы его можем получить и использовать в любой части приложения.

Расширения для добавления сервисов

Нередко для сервисов создают собственные методы добавления в виде методов расширения для интерфейса **IServiceCollection**. Например, создадим подобный метод для сервиса **TimeService**:

```
1 public static class ServiceProviderExtensions
2 {
3     public static void AddTimeService(this IServiceCollection services)
4     {
5         services.AddTransient<TimeService>();
6     }
7 }
```

И теперь используем этот метод:

```
1 var builder = WebApplication.CreateBuilder();
2
3 builder.Services.AddTimeService();
4
5 var app = builder.Build();
6 app.Run(async context =>
7 {
8     var timeService = app.Services.GetService<TimeService>();
9     context.Response.ContentType = "text/html; charset=utf-8";
10    await context.Response.WriteAsync($"Текущее время: {timeService?.GetTime()}");
11 });
12
13 app.Run();
14
15 public class TimeService
16 {
17     public string GetTime() => DateTime.Now.ToShortTimeString();
18 }
19
20 public static class ServiceProviderExtensions
21 {
22     public static void AddTimeService(this IServiceCollection services)
23     {
24         services.AddTransient<TimeService>();
25     }
26 }
```

3. Получение зависимостей

В ASP.NET Core мы можем получить добавленные в приложения сервисы различными **способами**:

Через свойство `Services` объекта `WebApplication` (service locator).

Через свойство `RequestServices` контекста запроса `HttpContext` в компонентах `middleware` (service locator).

Через конструктор класса.

Через параметр метода `Invoke` компонента `middleware`.

Через свойство `Services` объекта `WebApplicationBuilder`.

Для работы определим интерфейс **`ITimeService`** и класс **`ShortTimeService`**, который реализует данный интерфейс:

```
1 interface ITimeService
2 {
3     string GetTime();
4 }
5 class ShortTimeService : ITimeService
6 {
7     public string GetTime() => DateTime.Now.ToShortTimeString();
8 }
```

Свойство `Services` объекта `WebApplication`

Там, где нам доступен объект **`WebApplication`**, который представляет текущее приложение, (например, в файле `Program.cs`), для получения сервисов мы можем использовать его свойство **`Services`**. Это свойство предоставляет объект **`IServiceProvider`**, который предоставляет **ряд методов для получения сервисов**:

`GetService<service>()`: использует провайдер сервисов для создания объекта, который представляет тип **`service`**. В случае если в провайдере сервисов для данного сервиса не установлена зависимость, то возвращает значение **`null`**.

`GetRequiredService<service>()`: использует провайдер сервисов для создания объекта, который представляет тип **`service`**. В случае если в провайдере сервисов для данного сервиса не установлена зависимость, то генерирует исключение.

Данный паттерн получения сервиса еще называется **`service locator`**, и, как правило, не рекомендуется к использованию, но тем не менее в рамках ASP.NET Core в принципе мы можем использовать подобную функциональность особенно там, где другие способы получения зависимостей не доступны.

Например, определим следующий код приложения:

```

1 var builder = WebApplication.CreateBuilder();
2
3 builder.Services.AddTransient<ITimeService, ShortTimeService>();
4
5 var app = builder.Build();
6
7 app.Run(async context =>
8 {
9     var timeService = app.Services.GetService<ITimeService>();
10    await context.Response.WriteAsync($"Time: {timeService?.GetTime()}");
11 });
12
13 app.Run();
14
15 interface ITimeService
16 {
17     string GetTime();
18 }
19 class ShortTimeService : ITimeService
20 {
21     public string GetTime() => DateTime.Now.ToShortTimeString();
22 }

```

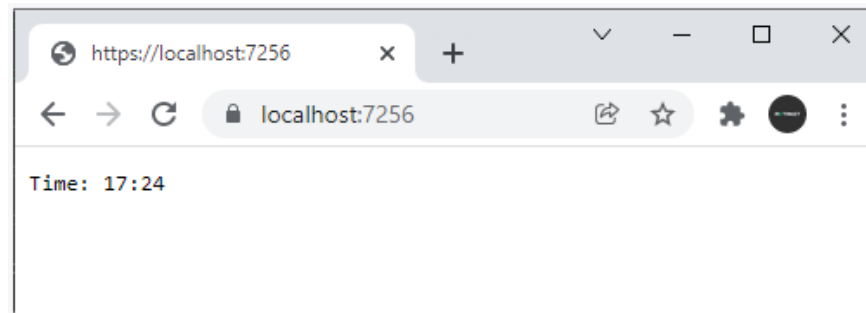
В данном случае с помощью строки кода

```

1 var timeService = app.Services.GetService<ITimeService>();

```

Получаем из коллекции сервисов объект сервиса **ITimeService** – в данном случае он будет представлять объект **ShortTimeService**.



Возможна ситуация, когда сервис не будет добавлен в коллекцию сервисов, однако в какой-то части приложения мы можем попытаться его получить:

```
1 var builder = WebApplication.CreateBuilder();
2
3 //builder.Services.AddTransient<ITimeService, ShortTimeService>();
4
5 var app = builder.Build();
6
7 app.Run(async context =>
8 {
9     var timeService = app.Services.GetService<ITimeService>();
10    await context.Response.WriteAsync($"Time: {timeService?.GetTime()}");
11 });
12
13 app.Run();
```

В этом случае переменная **timeService** будет иметь значение **null**.

Аналогичным образом можно использовать метод **GetRequiredService()** за тем исключением, что если сервис не добавлен, то метод генерирует исключение:

```
1 var builder = WebApplication.CreateBuilder();
2
3 builder.Services.AddTransient<ITimeService, ShortTimeService>();
4
5 var app = builder.Build();
6
7 app.Run(async context =>
8 {
9     var timeService = app.Services.GetRequiredService<ITimeService>();
10    await context.Response.WriteAsync($"Time: {timeService.GetTime()}");
11 });
12
13 app.Run();
```

Свойство RequestServices контекста запроса HttpContext.RequestServices

Там, где нам доступен объект **HttpContext**, мы можем использовать для получения сервисов его свойство **RequestServices**. Это свойство предоставляет объект **IServiceProvider**. То есть по сути мы имеем дело с выше описанным способом получения сервисов с помощью методов **GetService()** и **GetRequiredService()**:

```
1 var builder = WebApplication.CreateBuilder();
2
3 builder.Services.AddTransient<ITimeService, ShortTimeService>();
4
5 var app = builder.Build();
6
7 app.Run(async context =>
8 {
9     var timeService = context.RequestServices.GetService<ITimeService>();
10    await context.Response.WriteAsync($"Time: {timeService?.GetTime()}");
11 });
12
13 app.Run();
```

Конструкторы

Встроенная в ASP.NET Core система внедрения зависимостей использует конструкторы классов для передачи всех зависимостей. Передача сервисов через конструкторы является предпочтительным способом внедрения зависимостей.

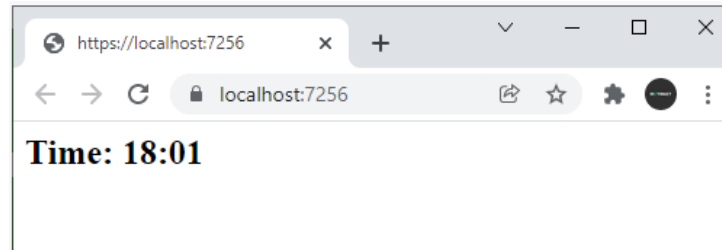
Например, пусть в проекте определен следующий класс **TimeMessage**:

```
1 class TimeMessage
2 {
3     ITimeService timeService;
4     public TimeMessage(ITimeService timeService)
5     {
6         this.timeService = timeService;
7     }
8     public string GetTime() => $"Time: {timeService.GetTime()}";
9 }
```


Здесь через конструктор класса передается зависимость от **ITimeService**. Причем здесь неизвестно, что это будет за реализация интерфейса **ITimeService**. В методе **GetTime()** формируем сообщение, в котором из сервиса получаем текущее время.

Для использования класса **TimeMessage** определим следующее приложение:

```
1  var builder = WebApplication.CreateBuilder();
2
3  builder.Services.AddTransient<ITimeService, ShortTimeService>();
4  builder.Services.AddTransient<TimeMessage>();
5
6  var app = builder.Build();
7
8  app.Run(async context =>
9  {
10     var timeMessage = context.RequestServices.GetService<TimeMessage>();
11     context.Response.ContentType = "text/html; charset=utf-8";
12     await context.Response.WriteAsync($"<h2>{timeMessage?.GetTime()}</h2>");
13 });
14
15 app.Run();
16
17 class TimeMessage
18 {
19     ITimeService timeService;
20     public TimeMessage(ITimeService timeService)
21     {
22         this.timeService = timeService;
23     }
24     public string GetTime() => $"Time: {timeService.GetTime()}";
25 }
26 interface ITimeService
27 {
28     string GetTime();
29 }
30 class ShortTimeService : ITimeService
31 {
32     public string GetTime() => DateTime.Now.ToShortTimeString();
33 }
```



Для использования в приложении в качестве сервиса класс **TimeMessage** также добавляется в коллекцию сервисов. Поскольку это самодостаточная зависимость, которая представляет конкретный класс, то метод **builder.Services.AddTransient** типизируется одним этим типом **TimeMessage**. То есть классы, которые используют сервисы, сами могут выступать в качестве сервисов.

Но так как класс **TimeMessage** использует зависимость **ITimeService**, которая передается через конструктор, то нам надо также установить и эту зависимость:

```
1 builder.Services.AddTransient<ITimeService, ShortTimeService>();
```

И когда при обработке запроса будет использоваться класс **TimeMessage**, для создания объекта этого класса будет вызываться провайдер сервисов. Провайдер сервисов проверят конструктор класса **TimeMessage** на наличие зависимостей. Затем создает объекты для всех используемых зависимостей и передает их в конструктор.

Метод **Invoke/InvokeAsync** компонентов **middleware**

Подобно тому, как зависимости передаются в конструктор классов, точно также их можно передавать в метод **Invoke/InvokeAsync()** компонента **middleware**. Например, определим следующий компонент:

```
1 class TimeMessageMiddleware
2 {
3     private readonly RequestDelegate next;
4
5     public TimeMessageMiddleware(RequestDelegate next)
6     {
7         this.next = next;
8     }
9
10    public async Task InvokeAsync(HttpContext context, ITimeService timeService)
11    {
12        context.Response.ContentType = "text/html; charset=utf-8";
13        await context.Response.WriteAsync($"<h1>Time: {timeService.GetTime()}</h1>");
14    }
15 }
```

Применим компонент для обработки запроса:

```
1 var builder = WebApplication.CreateBuilder();
2
3 builder.Services.AddTransient<ITimeService, ShortTimeService>();
4
5 var app = builder.Build();
6
7 app.UseMiddleware<TimeMessageMiddleware>();
8
9 app.Run();
10
11 class TimeMessageMiddleware
12 {
13     private readonly RequestDelegate next;
14
15     public TimeMessageMiddleware(RequestDelegate next)
16     {
17         this.next = next;
18     }
19
20     public async Task InvokeAsync(HttpContext context, ITimeService timeService)
21     {
22         context.Response.ContentType = "text/html; charset=utf-8";
23         await context.Response.WriteAsync($"<h1>Time: {timeService.GetTime()}</h1>");
24     }
25 }
26 interface ITimeService
27 {
28     string GetTime();
29 }
30 class ShortTimeService : ITimeService
31 {
32     public string GetTime() => DateTime.Now.ToShortTimeString();
33 }
```

Стоит отметить, что мы также могли бы передать зависимость и через конструктор класса **middleware**:

```
1 class TimeMessageMiddleware
2 {
3     RequestDelegate next;
4     ITimeService timeService;
5     public TimeMessageMiddleware(RequestDelegate next, ITimeService timeService)
6     {
7         this.next = next;
8         this.timeService = timeService;
9     }
10
11     public async Task InvokeAsync(HttpContext context)
12     {
13         context.Response.ContentType = "text/html; charset=utf-8";
14         await context.Response.WriteAsync($"<h1>Time: {timeService.GetTime()}</h1>");
15     }
16 }
```

Благодаря подобной передаче зависимости мы можем уйти от использования паттерна **"service locator"**, который демонстрировался в предыдущих примерах.

4. Жизненный цикл зависимостей

ASP.NET Core позволяет управлять жизненным циклом внедряемых в приложение сервисов. С точки зрения жизненного цикла сервисы могут представлять один из следующих типов:

Transient: при каждом обращении к сервису создается новый объект сервиса. В течение одного запроса может быть несколько обращений к сервису, соответственно при каждом обращении будет создаваться новый объект. Подобная модель жизненного цикла наиболее подходит для легковесных сервисов, которые не хранят данных о состоянии

Scoped: для каждого запроса создается свой объект сервиса. То есть если в течение одного запроса есть несколько обращений к одному сервису, то при всех этих обращениях будет использоваться один и тот же объект сервиса.

Singleton: объект сервиса создается при первом обращении к нему, все последующие запросы используют один и тот же ранее созданный объект сервиса

Для создания каждого типа сервиса предназначен соответствующий метод **AddTransient()**, **AddScoped()** и **AddSingleton()**. Для рассмотрения механизма внедрения зависимостей и жизненного цикла возьмем следующий интерфейс **ICounter**:

```
1 public interface ICounter
2 {
3     int Value { get; }
4 }
```

Также определим реализацию этого интерфейса – класс **RandomCounter**:

```
1 public class RandomCounter : ICounter
2 {
3     static Random rnd = new Random();
4     private int _value;
5     public RandomCounter()
6     {
7         _value = rnd.Next(0, 1000000);
8     }
9     public int Value
10    {
11        get => _value;
12    }
13 }
```

Суть класса **RandomCounter** состоит в генерации некоторого случайного числа в диапазоне от 0 до 1000000.

И также определим новый класс, который нам более детально поможет разобраться в механизме **Dependency Injection**. Этот класс назовем **CounterService** и определим в нем следующий код:

```
1 public class CounterService
2 {
3     public ICounter Counter { get; }
4     public CounterService(ICounter counter)
5     {
6         Counter = counter;
7     }
8 }
```

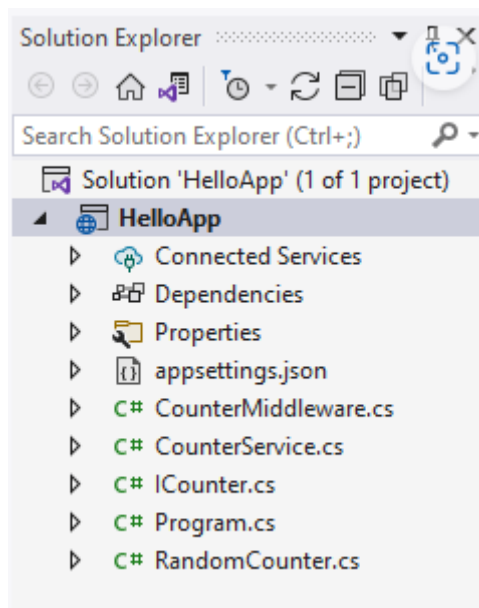
Данный класс просто устанавливает объект **ICounter**, передаваемый через конструктор.

Для работы с сервисами определим компонент **middleware**, который назовем **CounterMiddleware**:

```
1 public class CounterMiddleware
2 {
3     RequestDelegate next;
4     int i = 0; // счетчик запросов
5     public CounterMiddleware(RequestDelegate next)
6     {
7         this.next = next;
8     }
9     public async Task InvokeAsync(HttpContext httpContext, ICounter counter, CounterService counterService)
10    {
11        i++;
12        httpContext.Response.ContentType = "text/html;charset=utf-8";
13        await httpContext.Response.WriteAsync($"Запрос {i}; Counter: {counter.Value}; Service: {counterService.Counter.Value}");
14    }
15 }
```

Для получения зависимостей здесь используется метод **InvokeAsync**, в котором передаются две зависимости **ICounter** и **CounterService**. В самом методе выводятся значения **Value** из обеих зависимостей. Причем сервис **CounterService** сам использует зависимость **ICounter**.

То есть структура проекта будет выглядеть следующим образом:



Теперь на примере этих классов рассмотрим управление жизненным циклом сервисов.

AddTransient

Метод **AddTransient()** создает **transient**-объекты. Такие объекты создаются при каждом обращении к ним. Данный метод имеет ряд перегруженных версий:

AddTransient(Type serviceType).

AddTransient(Type serviceType, Type implementationType).

AddTransient(Type serviceType, Func<IServiceProvider,object> implementationFactory).

AddTransient<TService>().

AddTransient<TService, TImplementation>().

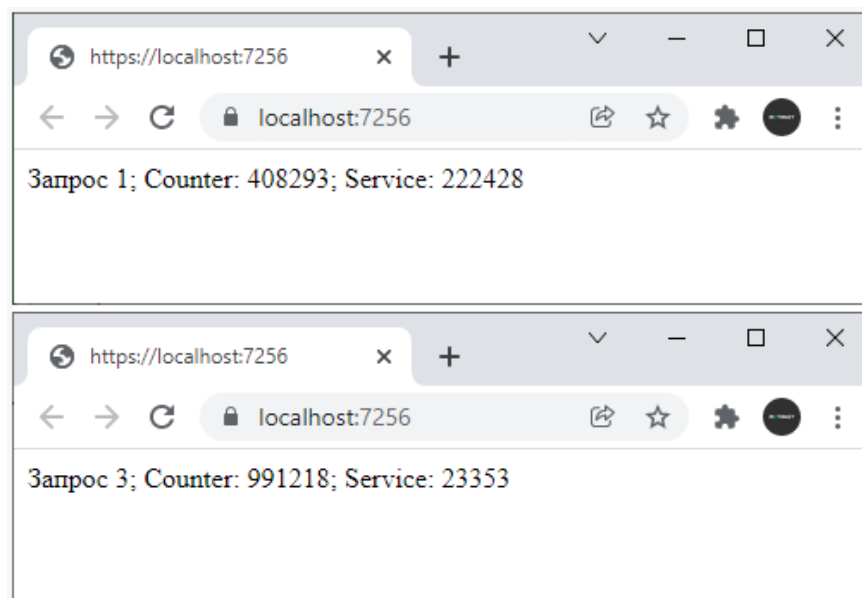
AddTransient<TService>(Func<IServiceProvider,TService> implementationFactory).

AddTransient<TService, TImplementation>(Func<IServiceProvider,TImplementation> implementationFactory).

Используем данный метод для добавления сервисов:

```
1 var builder = WebApplication.CreateBuilder();
2
3 builder.Services.AddTransient<ICounter, RandomCounter>();
4 builder.Services.AddTransient<CounterService>();
5
6 var app = builder.Build();
7
8 app.UseMiddleware<CounterMiddleware>();
9
10 app.Run();
```

Запустим проект:



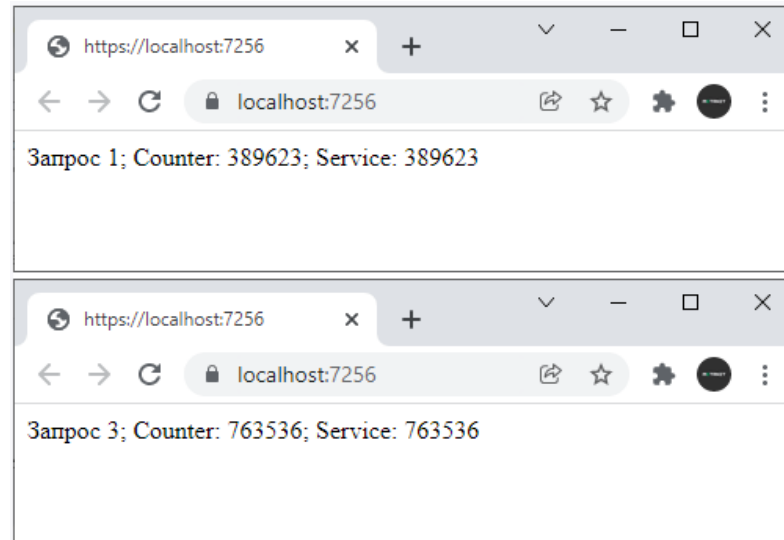
В нашем случае **CounterMiddleware** получает объект **ICounter**, для которого создается один экземпляр класса **RandomCounter**. **CounterMiddleware** также получает объект **CounterService**, который также использует **ICounter**. И для этого **ICounter** будет создаваться второй экземпляр класса **RandomCounter**. Поэтому генерируемые случайные числа обоими экземплярами не совпадают. Таким образом, применение **AddTransient** создаст два разных объекта **RandomCounter**.

При втором и последующих запросах к контроллеру будут создаваться новые объекты **RandomCounter**.

AddScoped

Метод **AddScoped** создает один экземпляр объекта для всего запроса. Он имеет те же перегруженные версии, что и **AddTransient**. Для его применения изменим код приложения следующим образом:

```
1 var builder = WebApplication.CreateBuilder();
2
3 builder.Services.AddScoped<ICounter, RandomCounter>();
4 builder.Services.AddScoped<CounterService>();
5
6 var app = builder.Build();
7
8 app.UseMiddleware<CounterMiddleware>();
9
10 app.Run();
```



Теперь в рамках одного и того же запроса и **CounterMiddleware** и сервис **CounterService** будут использовать один и тот же объект **RandomCounter**. При следующем запросе к приложению будет генерироваться новый объект **RandomCounter**.

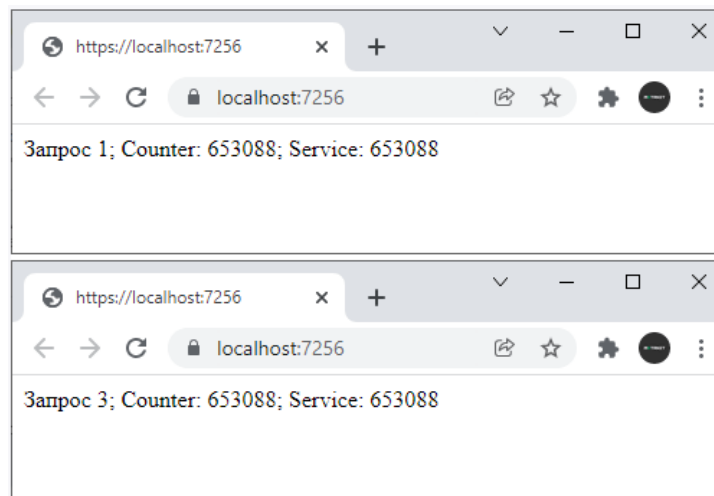
AddSingleton

AddSingleton создает один объект для всех последующих запросов, при этом объект создается только тогда, когда он непосредственно необходим. Этот метод имеет все те же перегруженные версии, что и **AddTransient** и **AddScoped**.

Для применения **AddSingleton** изменим код приложения:

```
1 var builder = WebApplication.CreateBuilder();
2
3 builder.Services.AddSingleton<ICounter, RandomCounter>();
4 builder.Services.AddSingleton<CounterService>();
5
6 var app = builder.Build();
7
8 app.UseMiddleware<CounterMiddleware>();
9
10 app.Run();
```

Как можно заметить, все три метода внедрения зависимостей имеют один и тот же принцип использования, различается только название метода. И в этом случае при нескольких последовательных запросах мы получим следующий результат:



Для создания **singleton**-объектов необязательно полагаться на механизм **Dependency Injection**. Мы его можем сами создать и передать в нужный метод:

```
1 var builder = WebApplication.CreateBuilder();
2
3 RandomCounter rndCounter = new RandomCounter();
4 builder.Services.AddSingleton<ICounter>(rndCounter);
5 builder.Services.AddSingleton<CounterService>(new CounterService(rndCounter));
6
7 var app = builder.Build();
8
9 app.UseMiddleware<CounterMiddleware>();
10
11 app.Run();
```

Работа приложения в данном случае будет аналогична предыдущему примеру.