

AddTransient, AddScoped and AddSingleton Services Differences

Asked 8 years, 5 months ago Modified 12 days ago Viewed 1.6m times



I want to implement [dependency injection](#) (DI) in ASP.NET Core. So after adding this code to `ConfigureServices` method, both ways work.

2159



What is the difference between the `services.AddTransient` and `service.AddScoped` methods in ASP.NET Core?



```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.

    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddScoped<IEmailSender, AuthMessageSender>();
}
```

c# asp.net-core .net-core

Share

Improve this question

Follow

edited Jun 8, 2021 at 0:10



boop

7,787 ● 14 ● 58 ● 99


asked Jul 1, 2016 at 6:03




Elvin Mammadov

27.3k ● 13 ● 44 ● 84

- 250 @tmg The docs say 'Transient lifetime services are created each time they are requested.' and 'Scoped lifetime services are created once per request.' which unless my grasp of English is weaker than I thought actually mean the exact same thing. – [Neutrino](#) Oct 5, 2017 at 15:20
- 128 @tmg I know. I'm just pointing out that the docs aren't at all clear on this point, so pointing people to the docs isn't very helpful. – [Neutrino](#) Oct 6, 2017 at 9:32
- 14 Late to the party, reading the comments even later, but I printed out that article, read it, and jotted the same observation in the margin that I now see @Neutrino made here. The article was ENTIRELY vague in offering that analysis. The example, thankfully, was less confusing. – [Wellspring](#) Dec 22, 2017 at 21:47
- 111 As far as I understand: *Transient lifetime services are created each time they are requested*. The word **requested** here is the everyday English meaning of asking for something, in this case a service. Whereas the word **request** in *once per request* refers to an HTTP Request. But I do understand the confusion. – [Memet Olsen](#) Aug 8, 2019 at 14:53

- 23 Transients means if `MyClass` needs an instance Of `SomeDependency` , it gets one, and if `SomeOtherClass` needs an instance, it gets a new instance of `SomeDependency` . Which is different from the instance given to `MyClass` . Scoped means (in the context of HTTP requests being handled) that an instance of `SomeDependency` , given to `MyClass` and `SomeOtherClass` will be the same (for the same HTTP request being handled).
- Mike de Klerk May 19, 2021 at 14:29 

14 Answers

Sorted by: Highest score (default) 



TL;DR

3419



Transient objects are always different; a new instance is provided to every controller and every service.

Scoped objects are the same within a request, but different across different requests.

Singleton objects are the same for every object and every request.

For more clarification, this example from [.NET documentation](#) shows the difference:

To demonstrate the difference between these lifetime and registration options, consider a simple interface that represents one or more tasks as an operation with a unique identifier, `operationId` . Depending on how we configure the lifetime for this service, the container will provide either the same or different instances of the service to the requesting class. To make it clear which lifetime is being requested, we will create one type per lifetime option:

```
using System;

namespace DependencyInjectionSample.Interfaces
{
    public interface IOperation
    {
        Guid OperationId { get; }
    }

    public interface IOperationTransient : IOperation
    {
    }

    public interface IOperationScoped : IOperation
    {
    }

    public interface IOperationSingleton : IOperation
    {
    }
}
```

```

public interface IOperationSingletonInstance : IOperation
{
}
}

```

We implement these interfaces using a single class, `Operation`, that accepts a GUID in its constructor, or uses a new GUID if none is provided:

```

using System;
using DependencyInjectionSample.Interfaces;
namespace DependencyInjectionSample.Classes
{
    public class Operation : IOperationTransient, IOperationScoped,
        IOperationSingleton, IOperationSingletonInstance
    {
        Guid _guid;
        public Operation() : this(Guid.NewGuid())
        {
        }

        public Operation(Guid guid)
        {
            _guid = guid;
        }

        public Guid OperationId => _guid;
    }
}

```

Next, in `ConfigureServices`, each type is added to the container according to its named lifetime:

```

services.AddTransient<IOperationTransient, Operation>();
services.AddScoped<IOperationScoped, Operation>();
services.AddSingleton<IOperationSingleton, Operation>();
services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));
services.AddTransient<OperationService, OperationService>();

```

Note that the `IOperationSingletonInstance` service is using a specific instance with a known ID of `Guid.Empty`, so it will be clear when this type is in use. We have also registered an `OperationService` that depends on each of the other `Operation` types, so that it will be clear within a request whether this service is getting the same instance as the controller, or a new one, for each operation type. All this service does is expose its dependencies as properties, so they can be displayed in the view.

```

using DependencyInjectionSample.Interfaces;

namespace DependencyInjectionSample.Services
{
    public class OperationService
    {

```

```

public IOperationTransient TransientOperation { get; }
public IOperationScoped ScopedOperation { get; }
public IOperationSingleton SingletonOperation { get; }
public IOperationSingletonInstance SingletonInstanceOperation { get; }

public OperationService(IOperationTransient transientOperation,
    IOperationScoped scopedOperation,
    IOperationSingleton singletonOperation,
    IOperationSingletonInstance instanceOperation)
{
    TransientOperation = transientOperation;
    ScopedOperation = scopedOperation;
    SingletonOperation = singletonOperation;
    SingletonInstanceOperation = instanceOperation;
}
}
}

```

To demonstrate the object lifetimes within and between separate individual requests to the application, the sample includes an `OperationsController` that requests each kind of `IOperation` type as well as an `OperationService`. The `Index` action then displays all of the controller's and service's `OperationId` values.

```

using DependencyInjectionSample.Interfaces;
using DependencyInjectionSample.Services;
using Microsoft.AspNetCore.Mvc;

namespace DependencyInjectionSample.Controllers
{
    public class OperationsController : Controller
    {
        private readonly OperationService _operationService;
        private readonly IOperationTransient _transientOperation;
        private readonly IOperationScoped _scopedOperation;
        private readonly IOperationSingleton _singletonOperation;
        private readonly IOperationSingletonInstance
        _singletonInstanceOperation;

        public OperationsController(OperationService operationService,
            IOperationTransient transientOperation,
            IOperationScoped scopedOperation,
            IOperationSingleton singletonOperation,
            IOperationSingletonInstance singletonInstanceOperation)
        {
            _operationService = operationService;
            _transientOperation = transientOperation;
            _scopedOperation = scopedOperation;
            _singletonOperation = singletonOperation;
            _singletonInstanceOperation = singletonInstanceOperation;
        }

        public IActionResult Index()
        {
            // ViewBag contains controller-requested services
            ViewBag.Transient = _transientOperation;
            ViewBag.Scoped = _scopedOperation;
            ViewBag.Singleton = _singletonOperation;
            ViewBag.SingletonInstance = _singletonInstanceOperation;
        }
    }
}

```

```

        // Operation service has its own requested services
        ViewBag.Service = _operationService;
        return View();
    }
}
}

```

Now two separate requests are made to this controller action:

DependencyInjectionSample Home Characters Register Log in

Lifetimes

Controller Operations

Transient	e6fee2c8-2122-4d10-aa05-cb376042e2c7
Scoped	661bff78-5ecb-4758-ae43-ec22a3e0babe
Singleton	93c52d5b-4c51-4907-a4d2-6a336a797ce6
Instance	00000000-0000-0000-0000-000000000000

OperationService Operations

Transient	a379336b-3fd0-49ac-b176-bae7c27c5de5
Scoped	661bff78-5ecb-4758-ae43-ec22a3e0babe
Singleton	93c52d5b-4c51-4907-a4d2-6a336a797ce6
Instance	00000000-0000-0000-0000-000000000000

Request One

DependencyInjectionSample Home Characters Register Log in

Lifetimes

Controller Operations

Transient	d0d9cf4c-9677-491e-a633-c6b961af938d
Scoped	2a801c7e-d9ac-41ac-8a4f-259e6ff0f92c
Singleton	93c52d5b-4c51-4907-a4d2-6a336a797ce6
Instance	00000000-0000-0000-0000-000000000000

OperationService Operations

Transient	11d7cfa8-e4e9-43e1-bb0e-b164a83854e2
Scoped	2a801c7e-d9ac-41ac-8a4f-259e6ff0f92c
Singleton	93c52d5b-4c51-4907-a4d2-6a336a797ce6
Instance	00000000-0000-0000-0000-000000000000

Request Two

Observe which of the `operationId` values varies within a request, and between requests.

- Transient objects are always different; a new instance is provided to every controller and every service.
- Scoped objects are the same within a request, but different across different requests
- Singleton objects are the same for every object and every request (regardless of whether an instance is provided in `ConfigureServices`)

Share

Improve this answer

Follow

edited Oct 29, 2020 at 15:03



David Pine

24.5k ● 11 ● 81 ● 113

answered Jul 1, 2016 at 7:27



akardon

45.8k ● 5 ● 35 ● 42

-
- 124 I understood the functions of each of them, but can someone explain the impact of using one instead of the other. What issues may it cause if not used correctly or choose one instead of another. – [pawan nepal](#) Aug 2, 2017 at 2:15
-
- 32 Say you are creating a request context related object (like the current user) with singleton scope then it's gonna remain the same instance across all the http requests which is not desired. IOC is all about creating instances, so we need to specify what's the scope of the created instance. – [akardon](#) Aug 2, 2017 at 14:58
-
- 6 Could you also explain the common pitfalls in which we nest transient or scoped dependencies in a singleton? – [Display Name](#) Jan 14, 2020 at 1:28
-
- 80 fair point! generally speaking if we put an object with a shorter lifetime in a longer living object, the IoC wouldn't create the inner object again. so say if you have a singleton which has a transient or scoped object in it, the inner object doesn't get recreated as the constructor of the singleton wouldn't get called again. but the other way around is OK. you can put a singleton in a transient with no problem. so the rule of thumb is the inner object should have an equal or longer lifetime than the outer one. – [akardon](#) Jan 16, 2020 at 11:01
-
- 2 @YaroslavTrofimov To be honest I didn't get the contradiction part, yeah that's true, each request creates a new instance of the controller and calls its pertinent method but the services that the controller's relaying on (the operations) could be in different scopes. which is fine. think of it this way, let's say in the controller method there is a loop in which each iteration needs to create a new instance of an object, if that object's lifetime is scoped it's gonna create the object once and reuse it multiple time, whereas if it was transient it would create multiple objects, – [akardon](#) May 12, 2020 at 7:41 ✎
-



578



In .NET's dependency injection there are three major lifetimes:

Singleton which creates a single instance throughout the application. It creates the instance for the first time and reuses the same object in the all calls.

Scoped lifetime services are created once per request within the scope. It is equivalent to a singleton in the current scope. For example, in MVC it creates one instance for each HTTP request, but it uses the same instance in the other calls within the same web request.

Transient lifetime services are created each time they are requested. This lifetime works best for lightweight, stateless services.

Here you can find and examples to see the difference:

[ASP.NET 5 MVC6 Dependency Injection in 6 Steps](#) (web archive link due to dead link)

[Your Dependency Injection ready ASP.NET : ASP.NET 5](#)

And this is the link to the official documentation:

[Dependency injection in ASP.NET Core](#)

Share

Improve this answer

Follow

edited May 5, 2020 at 8:00



Crypth

1,642 ● 19 ● 34

answered Jul 1, 2016 at 6:29



akardon

45.8k ● 5 ● 35 ● 42

-
- 48 Could you please explain why the Transient is the most lightweight? I thought the Transient is the most heavy work because it needs to create an instance every time for every injection.
– [Expert wanna be](#) Jul 20, 2016 at 10:22
-
- 36 You're right. Transient is not the most lightweight, I just said it's suitable for lightweight RESTful services :) – [akardon](#) Jul 20, 2016 at 22:40
-
- 6 So in which scenario we could use scoped and in which transient in controller example for example if we are retrieving few rows from database? I'm trying to understand scoped vs transient usage scenario in this case. – [sensei](#) Sep 24, 2017 at 19:18
-
- 16 it really depends on the logic you're expecting. For instance, if it's a single db call it actually doesn't make any difference which one you're using. but if you're calling db multiple times in the same request, then you can use scoped lifetime, as it keeps the same repository object in the memory and reuses multiple times within the same Http Request context. Whereas the transient one creates a new repository object multiple times (and consumes more memory). If you explain your specific scenario it'd be easy to judge which one suits better. – [akardon](#) Sep 25, 2017 at 1:13
-
- 14 One important point to highlight here is Singleton, Scoped and Transient are like russian dolls, one within the other. It is not possible to reverse their order when nesting, for eg. a scoped or singleton cannot be contained in a Transient, because we would be extending the lifetime of the parent which goes against containment! – [DL Narasimhan](#) Apr 21, 2020 at 19:20
-



Which one to use

516 Transient



- since they are created every time they will use **more memory** & Resources and can have a **negative** impact on performance



- use this for the **lightweight** service with little or **no state**.



Scoped

- better option when you want to maintain state within a request.

Singleton

- memory leaks in these services will build up over time.
- also memory efficient as they are created once reused everywhere.

Use Singletons where you need to maintain application wide state. Application configuration or parameters, Logging Service, caching of data is some of the examples where you can use singletons.

Injecting service with different lifetimes into another

1. **Never inject Scoped & Transient services into Singleton service.** (This effectively converts the transient or scoped service into the singleton.)
2. **Never inject Transient services into scoped service** (This converts the transient service into the scoped.)

Note: I think it's fair to say that the above advice is seriously disputed. Many developers think it's fine to inject, for example, a transient into a singleton.

Share

Improve this answer

Follow

edited Apr 21, 2023 at 12:01

answered May 11, 2020 at 9:15



Jez

29.8k ● 36 ● 152 ● 251



bereket gebredingle

13k ● 3 ● 40 ● 49

-
- 19 I don't understand why **Transient** is recommended for "lightweight service with little or no state". Why not singleton in such case? Wouldn't it be even better to instantiate that small service just once and use it multiple times since it's stateless? Even if the service instantiation is cheap, if you do it a lot of times, the overhead will grow. With singleton, it stays the same
– [mnj](#) Jul 8, 2020 at 8:00
-
- 5 It should be added that when using singletons, you must make sure they are thread-safe since they can be used by multiple concurrent requests running on separate threads.
– [Eric Mutta](#) Jan 2, 2021 at 14:56
-
- 12 What's the issue with injecting a transient service into a scoped service? From my understanding, doing so does not make the transient services *become* a singleton (if you injected the same transient service somewhere else, it would be a different object), so provided the transient service has no state (which should be implicit), I don't see a problem.
– [ajbeaven](#) Jan 12, 2021 at 3:55
-

5 @S-eagle Could you give an example of such a stateless class that would gain some performance if it is instantiated per request (transient)? I'd really like to gain a good understanding of this. – [mnj](#) Jan 23, 2021 at 19:44

3 If you are not supposed to inject transient services into neither scoped nor singleton, then where would you inject your transient services? – [Snippy Valson](#) Aug 9, 2021 at 12:31

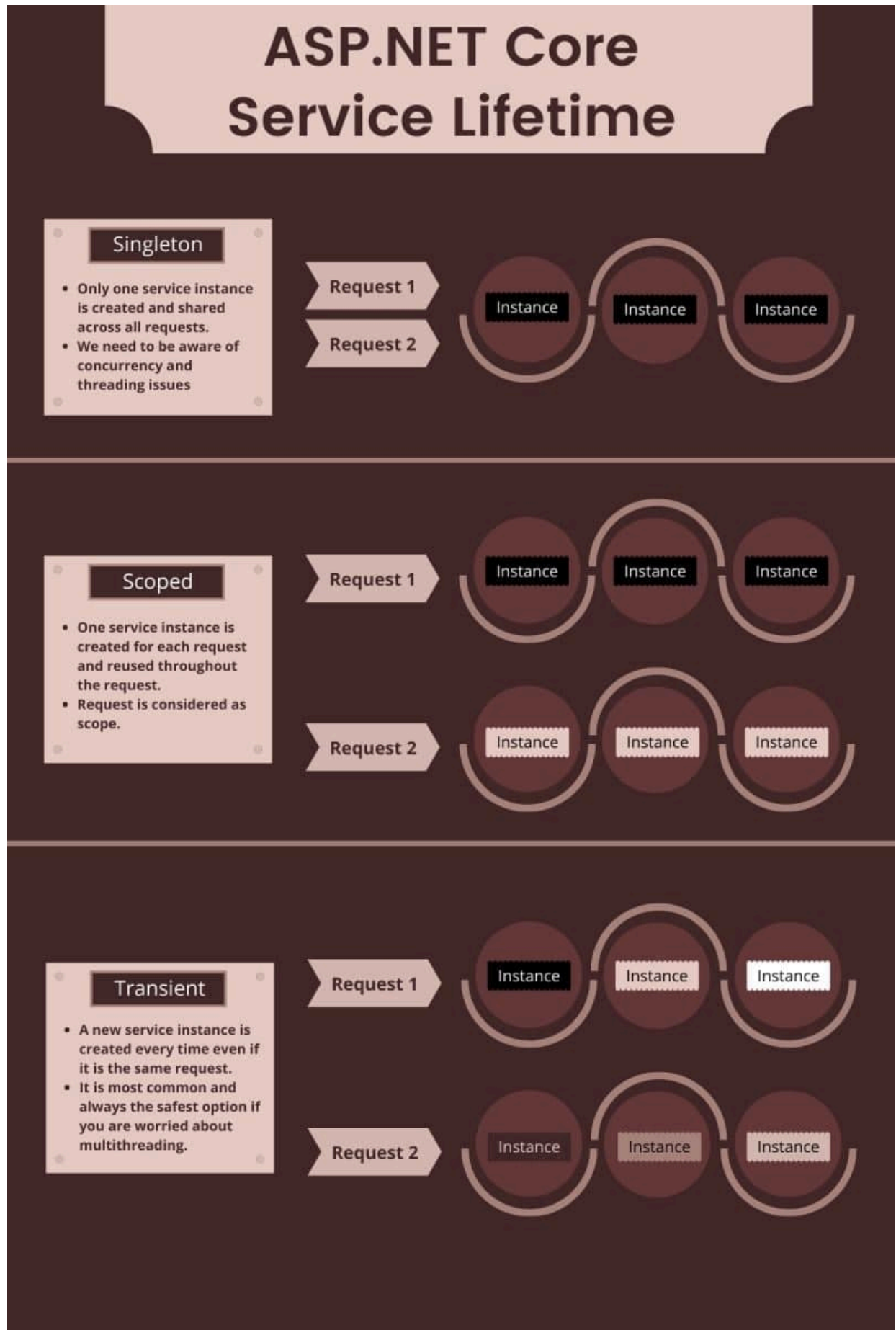


This image illustrates this concept well. Unfortunately, I could not find the source of this image, but someone made it, he has shown this concept very well in the form of

184



an image.



Update: Image reference : [ASP.NET Core Service Lifetimes \(Infographic\)](#) , Author: [@WaqasAnwar](#)

36 Here is the original source of the above image. ezylearning.net/tutorial/... Actually I posted it on my blog 5 days ago :-) – [Waqas Anwar](#) Nov 13, 2020 at 17:43 ✎

I read your article and I have a ton of these `services.AddTransient<IProductService, ProductService>()`; . I have a service that has a count of 193 in memory! This service just has stateless methods, should this be scoped instead of transient so I can have only one created for all my controllers? – [Mike Flynn](#) May 12, 2021 at 21:19 ✎

@MikeFlynn For one instance per each request you should use `AddScoped<IProductService, ProductService>()`; . but for one instance for all of requests use `AddSingleton<IProductService, ProductService>()`; – [Hamed Naeemaei](#) May 15, 2021 at 4:43

1 Wouldn't the singleton be held in memory until an application restart happens? I don't want a ton of singletons hanging around. – [Mike Flynn](#) May 15, 2021 at 11:34

Yes remains in memory until the restart – [Hamed Naeemaei](#) Jan 26, 2023 at 10:35



79



Transient, scoped and singleton define object creation process in ASP.NET MVC core DI(Dependency Injection) when multiple objects of the same type have to be injected. In case you are new to dependency injection you can see this [DI IoC video](#).

You can see the below controller code in which I have requested two instances of "IDal" in the constructor. **Transient, Scoped and Singleton** define if the same instance will be injected in "_dal" and "_dal1" or different.

```
public class CustomerController : Controller
{
    IDal dal = null;

    public CustomerController(IDal _dal,
                             IDal _dal1)
    {
        dal = _dal;
        // DI of MVC core
        // inversion of control
    }
}
```

Transient: In transient, new object instances will be injected in a single request and response. Below is a snapshot image where I displayed GUID values.

```

((CustomerApplication.Dal.AdoDal)_dal).Guid  P - "c52b4baf-38c7-4b65-a5c0-56594c58780f"
public class CustomerController : Controller
{
    ((CustomerApplication.Dal.AdoDal)_dal1).Guid  P - "e9af3f1d-6930-4f6f-ab15-5bcb59519bd4"
    IDal _dal = null;
    public CustomerController(IDal _dal
                            ,IDal _dal1)
    {

```

Different Instance injected

Scoped: In scoped, the same object instance will be injected in a single request and response.

```

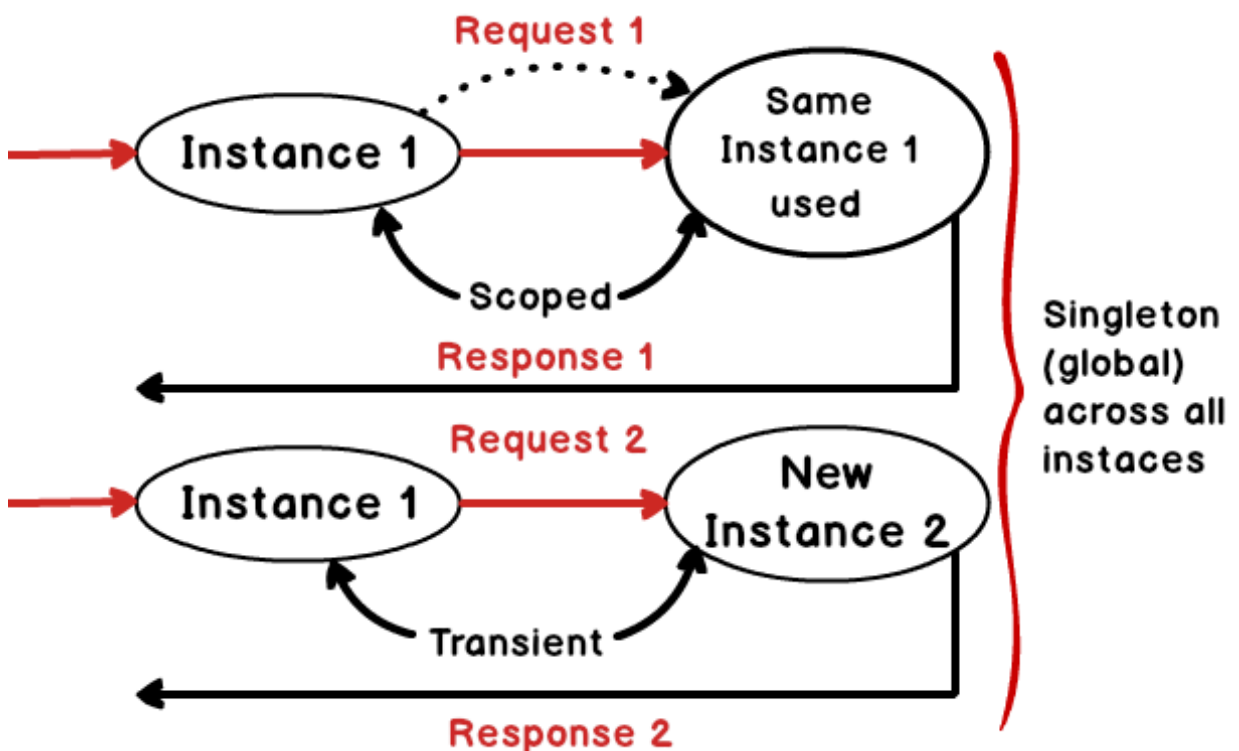
((CustomerApplication.Dal.AdoDal)_dal).Guid  P - "93d5ca90-c301-404d-9e0a-daa4d611effb"
((CustomerApplication.Dal.AdoDal)_dal1).Guid  P - "93d5ca90-c301-404d-9e0a-daa4d611effb"
public CustomerController(IDal _dal
                        ,IDal _dal1)
{
    dal = _dal;
    // DI of MVC core
    // inversion of control
}

```

Same Instance injected

Singleton: In singleton, the same object will be injected across all requests and responses. In this case one global instance of the object will be created.

Below is a simple diagram which explains the above fundamental visually.



The above image was drawn by the SBSS team when I was taking [ASP.NET MVC training in Mumbai](#). A big thanks goes to the SBSS team for creating the above image.

Share

Improve this answer

Follow

edited Jul 19, 2021 at 1:56



Dhana

1,658 ● 4 ● 24 ● 40

answered Nov 5, 2017 at 3:55



Shivprasad Koirala

28.5k ● 7 ● 87 ● 75

26 This is the single most complicated explanation of a transient service I've ever seen. Transient = Any time this service is resolved is the equivalent of assigning your variable `new TService`. Scoped will cache the first initialisation of it for that "scope" (http request in most cases). Singleton will cache only one instance ever for the lifetime of the application, Simple as that. The above diagrams are so convoluted. – [Mardox](#) Mar 2, 2018 at 16:28 ✎

6 So sorry i thought i will make it more simpler with diagrams and code snapshot :-) But i do get your point. – [Shivprasad Koirala](#) Mar 4, 2018 at 2:14

1 I found this helpful in the unique case where you have multiple instances injected, and Transient registration is used. Thanks – [Stokely](#) Nov 16, 2020 at 20:27

What scope does the controller itself have? – [variable](#) Jul 13, 2022 at 13:04

1 @DavidKlempfner Anywhere where you need to have new instances created per "request"; whatever a request is. E.g. a worker which listens for files (meaning any datastream) to process. You could have a scoped container for each processing task and resolve from that, rather than chaining dependencies along or making all your instances singletons. – [Mardox](#) May 12, 2023 at 23:06



75



AddSingleton()

AddSingleton() creates a single instance of the service when it is first requested and reuses that same instance in all the places where that service is needed.

AddScoped()

In a scoped service, with every HTTP request, we get a new instance. However, within the same HTTP request, if the service is required in multiple places, like in the view and in the controller, then the same instance is provided for the entire scope of that HTTP request. But every new HTTP request will get a new instance of the service.

AddTransient()

With a transient service, a new instance is provided every time a service instance is requested whether it is in the scope of the same HTTP request or across different HTTP requests.

Share

Improve this answer

Follow

edited Jun 20, 2020 at 9:12



Community Bot

1 ● 1

answered May 16, 2019 at 6:33



Yasser Shaikh

47.7k ● 49 ● 207 ● 286

-
- 1 Does scoped lifetime have any meaning outside of ASP.NET, for example in a .NET 6 worker service application? – C-F Aug 3, 2022 at 22:02
-

@C-F, It does have meaning, but you create your scopes by yourself when you need it. See [learn.microsoft.com/en-us/dotnet/core/extensions/...](https://learn.microsoft.com/en-us/dotnet/core/extensions/) – C-F Aug 3, 2022 at 22:08

▲

59

▼

- Singleton is a single instance for the lifetime of the application domain.
- Scoped is a single instance for the duration of the scoped request, which means per *HTTP* request in ASP.NET.
- Transient is a single instance per *code* request.

Normally the code request should be made through a constructor parameter, as in

```
public MyConsumingClass(IDependency dependency)
```

I wanted to point out in @akazemis's answer that "services" in the context of DI does not imply RESTful services; services are implementations of dependencies that provide functionality.

Share Improve this answer Follow

answered Nov 8, 2017 at 19:43



user1969177

786 ● 5 ● 8

▲

40

▼

After looking for an answer for this question I found a brilliant explanation with an example that I would like to share with you.

You can watch a video that demonstrate the differences [HERE](#)

In this example we have this given code:

```
public interface IEmployeeRepository
{
    IEnumerable<Employee> GetAllEmployees();
    Employee Add(Employee employee);
}

public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
}

public class MockEmployeeRepository : IEmployeeRepository
{
    private List<Employee> _employeeList;
```

```

public MockEmployeeRepository()
{
    _employeeList = new List<Employee>()
    {
        new Employee() { Id = 1, Name = "Mary" },
        new Employee() { Id = 2, Name = "John" },
        new Employee() { Id = 3, Name = "Sam" },
    };
}

public Employee Add(Employee employee)
{
    employee.Id = _employeeList.Max(e => e.Id) + 1;
    _employeeList.Add(employee);
    return employee;
}

public IEnumerable<Employee> GetAllEmployees()
{
    return _employeeList;
}
}

```

HomeController

```

public class HomeController : Controller
{
    private IEmployeeRepository _employeeRepository;

    public HomeController(IEmployeeRepository employeeRepository)
    {
        _employeeRepository = employeeRepository;
    }

    [HttpGet]
    public ActionResult Create()
    {
        return View();
    }

    [HttpPost]
    public ActionResult Create(Employee employee)
    {
        if (ModelState.IsValid)
        {
            Employee newEmployee = _employeeRepository.Add(employee);
        }

        return View();
    }
}

```

Create View

```

@model Employee
@inject IEmployeeRepository empRepository

```



```
<form asp-controller="home" asp-action="create" method="post">
  <div>
    <label asp-for="Name"></label>
    <div>
      <input asp-for="Name">
    </div>
  </div>

  <div>
    <button type="submit">Create</button>
  </div>

  <div>
    Total Employees Count =
    @empRepository.GetAllEmployees().Count().ToString()
  </div>
</form>
```

Startup.cs

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddSingleton<IEmployeeRepository, MockEmployeeRepository>();
}
```

Copy-paste this code and press on the create button in the view and switch between `AddSingleton` , `AddScoped` and `AddTransient` you will get each time a different result that will might help you understand this.

AddSingleton() - As the name implies, `AddSingleton()` method creates a Singleton service. A Singleton service is created when it is first requested. This same instance is then used by all the subsequent requests. So in general, a Singleton service is created only one time per application and that single instance is used throughout the application life time.

AddTransient() - This method creates a Transient service. A new instance of a Transient service is created each time it is requested.

AddScoped() - This method creates a Scoped service. A new instance of a Scoped service is created once per request within the scope. For example, in a web application it creates 1 instance per each http request but uses the same instance in the other calls within that same web request.

Share

edited Oct 3, 2020 at 10:56

answered Jul 11, 2019 at 16:19

Improve this answer

Follow



Offir

3,471 ● 4 ● 48 ● 79

Adding to all the fantastic answers here. This answer will explain the different lifetime and when it is appropriate to choose them. As with many things in software development, there are no absolutes and many condition can affect the best choices, so please treat this answer as a general guidance.

ASP.NET Core ships with its own built-in Dependency Injection Container which it uses to resolve the services it needs during the request lifecycle. All framework services, for example, like Logging, configuration, Routing etc., use dependency injection and they are registered with the Dependency Injection Container when the application host is built. Internally, the ASP.NET Core framework provides the dependencies required when activating framework components, such as Controllers and Razor pages.

A Dependency Injection Container, sometimes referred to as an Inversion of Control, or IoC Container, is a software component that manages the instantiation and configuration of objects. The dependency Injection Container is not a requirement to apply the dependency injection pattern, but as your application grows using one can greatly simplify the management of dependencies, including their lifetimes. Services are registered with the container at start up and resolved from the container at runtime whenever they are required. The container is responsible for creating and disposing of instances of required services, maintaining them for a specified lifetime.

There are two primary interfaces that we code against when using the Microsoft Dependency Injection Container.

1. The `IServiceCollection` interface defines a contract for registering and configuring a collection of service descriptors. From the `IServiceCollection` we build an `IServiceProvider`.
2. `IServiceProvider`, defines a mechanism for resolving a service at runtime.

When registering a service with the container, a service lifetime should be chosen for the service. The service lifetime controls how long a resolved object will live for after the container has created it. The lifetime can be defined by using the appropriate extension method on the `IServiceCollection` when registering the service. There are three lifetimes available for use with the Microsoft Dependency Injection Container.

1. Transient
2. Singleton
3. Scoped

The Dependency Injection Container keeps track of all the instances of services it creates, and they are disposed of or released for garbage collection once their lifetime has ended. The chosen lifetime affects whether the same service instance may be

resolved and injected into more than one dependent consumer. It is crucial, therefore, to choose the lifetime of services very wisely.

Transient Services

When a service is registered as Transient, a new instance of that service is created and returned by the container every time the service is resolved. In other words, every dependent class that accepts a Transient service via injection from the container will receive its own unique instance. Because each dependent class receives its own instance, methods on the instance are safe to mutate internal state without fear of access by other consumers and threads.

Uses/ Characteristics of Transient Services

1. Transient service is most useful when the service contains mutable state and is not considered thread safe.
2. Using Transient services can come with a small performance cost, although that cost may be negligible for applications under general load.
3. Whenever an instance is resolved, which could be as often as every request, memory for that instance must be allocated. This leads to additional work for the garbage collector in order to clean up those short lived objects.
4. Transient service are easiest to reason about because instances are never shared; therefore they tend to be safest choice when it is unclear which lifetime is going to be the best option when you register a service.

Singleton Services

An application service registered with the singleton lifetime will be created only once during the lifetime of the Dependency Injection Container. In ASP.NET Core, this is equivalent to the lifetime of the application. The first time the service is required, the container will create an instance. After that, the same instance can be reused and injected into all dependent classes. The instance will remain reachable for the lifetime of the container, so it does not require disposal or garbage collection.

Uses/ Characteristics of Singleton Service

1. Suppose the service is required frequently, such as per request. This can improve application performance by avoiding repeated allocation of new objects, each of which may require garbage collection only moments later.
2. Additionally, if the object is expensive to construct, limiting this to a single instance can improve your application performance, as it only takes place one time when the service is first used.

3. When choosing to register a service with Singleton lifetime, you must consider thread safety. Because the same instance of a Singleton service can be used by multiple requests concurrently. Any mutable state without suitable locking mechanism could lead to unexpected behaviour.
4. Singleton are very well suited to functional-style services, where the methods take an input and return an output, with no shared state being used.
5. A reasonable use case for Singleton lifetime in ASP.NET Core is for the memory cache, where the state must be shared for the cache to function.
6. When registering services for Singleton, consider the implications of a single instance remaining allocated for the life of the application.
 - a. It is possible to create memory leaks if the instance holds large amount of memory
 - b. This can be especially problematic if memory usage can grow during the instance's lifetime, given that it will never be released for garbage collection.
 - c. If a service has high memory requirements but is used very infrequently, then the singleton lifetime may not be the most appropriate choice.

Scoped Services

Scoped services sit in a middle ground between Transient and Singleton. An Instance of the scoped service lives for the length of the scope from which it is resolved. In ASP.NET Core, a scope is created within your application for each request it handles. Any scope services will be created once per scope, so that they act similarly to singleton services, but within the context of the scope. All framework components such as middleware and MVC controllers get the same instance of a scoped service when handling a particular request.

Uses/ Characteristics of Scoped Services

1. The container creates a new instance per request.
2. Because the container resolves a new instance of the type per request, it is not generally required to be thread safe.
3. Components within the request life cycle are invoked in sequence, so the shared instance is not used concurrently.
4. Scoped services are useful if multiple consumers may require the same dependency during a request.
 - a. An excellent example of such a case is when using Entity Framework Core. By default, the DbContext is registered with the scoped lifetime. Change tracking, therefore, applies across the entire request. Multiple components can make changes to the shared DbContext.

Avoiding Captive Dependencies

When registering your dependencies, it is crucial to ensure that the chosen lifetime is appropriate considering any dependencies that the service has of its own. This is necessary to avoid something called captive dependencies, which is where a service may live longer than is intended.

The thumb rule is, a service should not depend on a service with a lifetime shorter than its own. For example, a service registered with the singleton lifetime should not depend on a transient service. Doing so would result in the transient service being captured by the singleton service, with the instance unintentionally being referenced for the life of the application. This can lead to problematic and sometime hard to track down runtime bugs and behaviours, such as accidentally sharing non-thread safe services between threads or allowing objects to live past their intended lifetime. To visualize this, let us consider which lifetimes can safely depend on services using another lifetime.

1. Since it will be a short-lived service, a transient service can safely depend on service that have transient, scoped or singleton lifetime
2. Scope services are little tricky. If they depend on a transient service, then a single instance of that transient service will live for the life of the scope across the whole request. You may or may not want this behaviour. To be absolutely safe, you might choose not to depend on transient services from scoped services, but it is safe for a scoped service to depend on other scoped or singleton services.
3. A singleton service is most restrictive in terms of its dependencies, and it should not depend on transient or scoped services, but can depend upon other singleton services. Capture of scope services by singletons is one of the more dangerous possibilities. Because scoped services could be disposed of then the scope ends, It is possible that the singleton may try and access them after their disposal. This could lead to runtime exceptions in production, a really bad situation.

Safe Dependencies

	Transient	Scoped	Singleton
Transient	✓	✓	✓
Scoped	✗	✓	✓
Singleton	✗	✗	✓

Share

Improve this answer

Follow

edited Oct 11, 2023 at 9:38



Petter Hesselberg

5,478 ● 3 ● 27 ● 51

answered Jul 7, 2023 at 7:42



j4jada

662 ● 7 ● 12

Are services = dependencies in .Net ? I am confused with the concept of services in .Net compared to Angular or Laravel. – [CloudWave](#) Jan 22 at 10:12

ASP.NET Core includes dependency injection (DI) that makes configured services available throughout an app. Services (e.g. logging, routing etc..) are typically resolved from DI using constructor injection. The DI framework provides an instance of this service at runtime.

– [j4jada](#) Jan 22 at 23:26

- 1 among the answers, this one explains clearly what is the role of each type of DI, potential use cases, and accompanied issues such as memory leak and captive dependencies. The problem with other answers are to explain the behaviour of each type, which already clear from Microsoft documentation. – [hakuna1811](#) Sep 26 at 23:34



In `.NET Core` all of these techniques `AddTransient`, `AddScoped` and `AddSingleton` define the lifetime of injected object.

19



AddSingleton - Only one global instance of the class throughout the application will be created.



```
builder.Services.AddSingleton<ConsoleLogger>();
```



AddScoped - For every single request different instance will be created **but** within a same request if you are asking dependency to inject object two times, three times, same instance will be served, all objects will be identical. See example below

```
builder.Services.AddScoped<ConsoleLogger>();
```

Now in the Controller class

```
ConsoleLogger _logger;  
public HomeController( ConsoleLogger logger, ----- Same object  
                      ConsoleLogger logger1) ----- Same object  
{  
    _logger = consoleLogger;  
}
```

Here `logger` is equal to `logger1`.

AddTransient - For every request new instance will be created **even if** within the request if you ask DI to create object multiple times, every time new instance will be created. See example below

```
builder.Services.AddTransient<ConsoleLogger>();
```

Now in the Controller class

```
ConsoleLogger _logger;  
public HomeController( ConsoleLogger logger, ----- New object  
                      ConsoleLogger logger1) ----- New object  
{  
    _logger = consoleLogger;  
}
```

Here `logger` is not equal to `logger1`. hope it helps.

Share

edited Dec 9 at 5:33

answered Dec 14, 2023 at 17:11

Improve this answer

Follow



Arvind Chourasiya

17.3k ● 15 ● 119 ● 202

- 1 when the Singleton object will be created, while server started, or when server receives first request? – [logeshpalani31](#) Apr 9 at 7:41

@logeshpalani31 - If you register a service as a Singleton, it will be created when the DI container is built, typically during application startup. This means it's created when the server starts. – [Arvind Chourasiya](#) Apr 28 at 15:41



DI containers can be pretty mystifying at first, especially with regard to lifetimes. After all, containers use reflection to make everything "just work." It helps to think about



what containers are actually accomplishing for you under the hood: composing object graphs.



For a .NET web app, the alternative to using a DI container is to replace the default controller activator with your own, which must manage lifetimes and construct dependency graphs manually. For learning purposes, pretend you have a controller activator that is hard-coded to return one particular controller each time there is a web request:

```
// This class is created once per application during startup. In DI terms, it
// is the
// "composition root."
public class DumbControllerActivator
{
    // Shared among all consumers from all requests
    private static readonly Singleton1 singleton1 = new Singleton1();
    private static readonly Singleton2 singleton2 = new Singleton2();

    // This method's responsibility is to construct a FooController and its
    // dependencies.
    public FooController HandleFooRequest()
    {
        // Shared among all consumers in this request
        var scoped1 = new Scoped1();
        var scoped2 = new Scoped2(singleton1, scoped1);

        return new FooController(
            singleton1,
            scoped1,
            new Transient1(                // Fresh instance
                singleton2,
                new Transient2(scoped2)), // Fresh instance
            new Transient3(                // Fresh instance
                singleton1,
                scoped1,
                new Transient1(            // Fresh instance
                    singleton2,
                    new Transient2(scoped2))); // Fresh instance
        );
    }
}
```

- The activator creates each singleton instance only once and then holds onto it throughout the lifetime of the application. Each consumer shares that single instance (even consumers from separate requests).
- For scoped dependencies, the activator creates one instance per web request. Within that request, every consumer shares that single instance, but from request to request, the instances are different.
- For transient dependencies, each consumer gets its own private instance. There is no sharing at all.

For a much deeper dive into DI, I highly recommend the book [Dependency Injection Principles, Practices, and Patterns](#). My answer is basically just repeating what I learned there.

Share

Improve this answer

Follow

edited May 28, 2022 at 6:53



codevision

5,510 ● 41 ● 54

answered Feb 19, 2022 at 21:13



MarredCheese

20.7k ● 11 ● 106 ● 103



13



There are fantastic answers. I want to give a brief idea with analogy.

Imagine you are running a coffee shop, and you have three types of employees: baristas, cashiers, and managers. Each type of employee represents a different dependency with a different lifetime in your coffee shop application.

1. Transient Lifetime (Baristas):

- Transient services are like hiring a new barista for each customer who walks into your coffee shop.
- When a customer comes in, you hire a new barista to make their coffee. After the customer leaves, that barista is no longer needed and can be let go.

```
services.AddTransient<IBarista, Barista>();
```

2. Scoped Lifetime (Cashiers):

- Scoped services are like hiring a cashier for a single shift in your coffee shop.
- When a customer comes in during a shift, the same cashier handles all their transactions. Once the shift ends, the cashier can go home, and a new cashier is hired for the next shift.

```
services.AddScoped<ICashier, Cashier>();
```

3. Singleton Lifetime (Manager):

- Singleton services are like having a single manager overseeing your coffee shop at all times.
- The manager is responsible for managing everything in the coffee shop and remains in place as long as the coffee shop is open.

```
services.AddSingleton<IManager, Manager>();
```

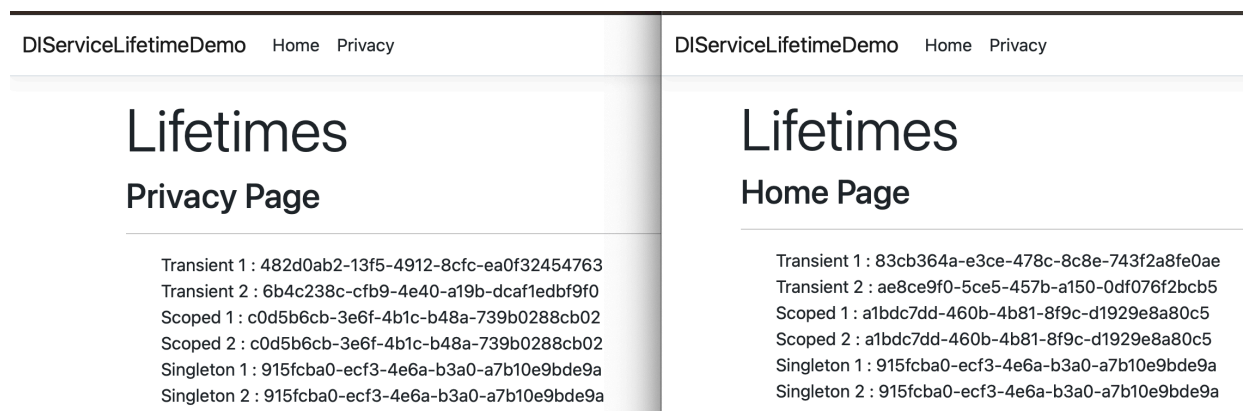
Now, let's apply this analogy to retrieving items from a database:

- **Transient Lifetime (Baristas):** If you configure a service as transient, a new instance of that service is created every time it's needed. In the context of a database, a new database connection or data access component is created for each database query. After the query is complete, that connection is closed and disposed of.
- **Scoped Lifetime (Cashiers):** If you configure a service as scoped, a single instance of that service is created for the duration of an operation or request. In the database context, a single database connection or data access component is created and shared across all queries within the same operation or request. Once the operation or request is complete, the connection is closed and disposed of.
- **Singleton Lifetime (Manager):** If you configure a service as a singleton, only one instance of that service is created and shared across the entire application's lifetime. In the database context, a single database connection or data access component is created and used for all queries from the start of the application until it's shut down.

The choice of which lifetime to use depends on factors like resource efficiency, isolation requirements, and concurrency considerations in your specific application scenario. Different lifetimes provide different trade-offs, and it's essential to choose the one that aligns with your application's requirements and performance constraints.

In addition to that information, I have built a web project to test them based on the above fantastic information given by other users.

That's the result of the web application:



As you can see in the picture, two different requests, **Index** and **Privacy**, return different results for the *transient* and *scoped*.

If anyone wonders the codes, this is the application I've created.

[The source code in the GitHub](#)

Share

Improve this answer

Follow

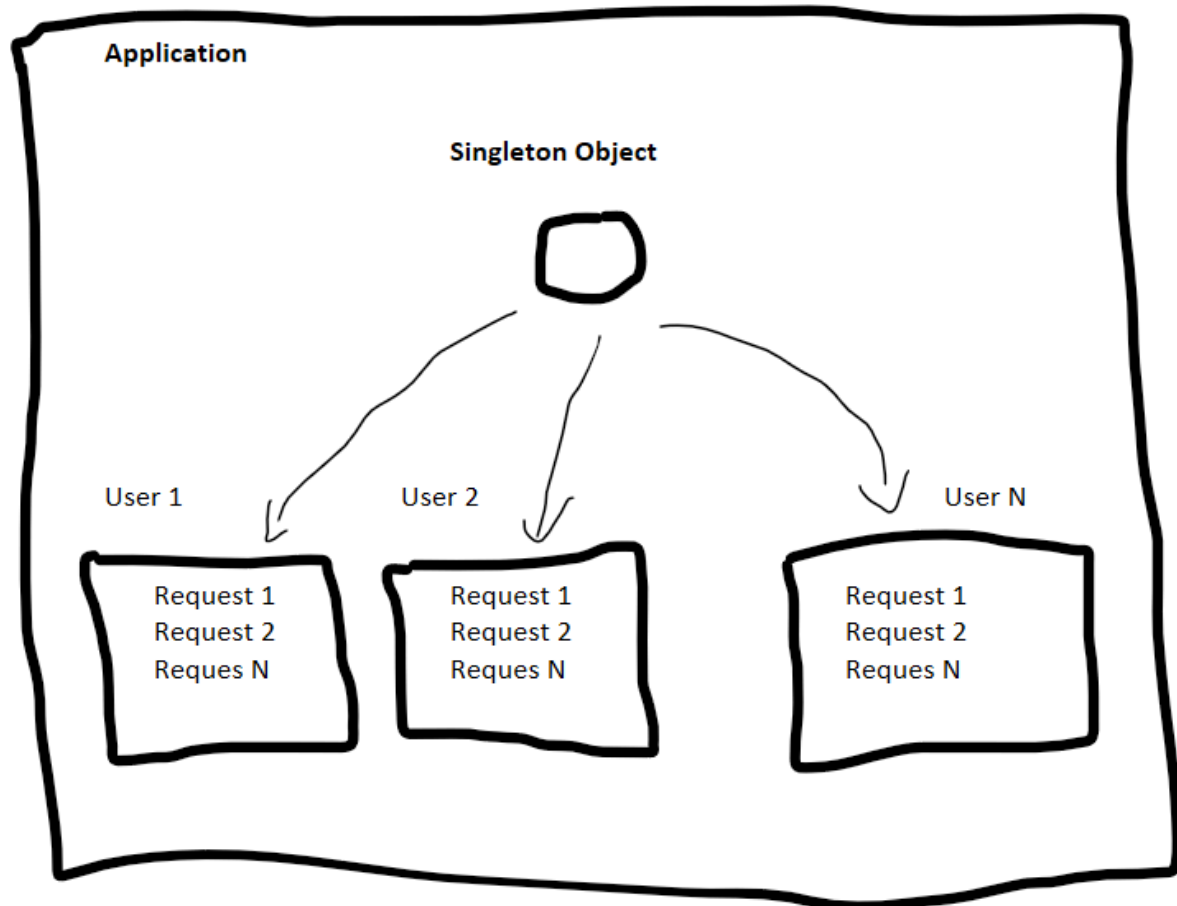


Cem Kaya

167 ● 1 ● 5

- 1 did you use any AI at all in the writing of any part of this answer? – [starball](#) Sep 19, 2023 at 7:19

I like to draw what I understand.



User N - Request N

Controller

Scoped Object



Constructor(IScopedObject obj, IOtherObject otherObj1, IOtherObject otherObj2,...)

- If the constructor have the Scoped object injected then 1 instance will be created.
- Any object injected in the constructor that have the Scoped Object injected in its constructor or other method, will use the same scoped object in the same request.

otherObj1

Constructor(IScopedObject obj)

otherObj2

Constructor(IScopedObject obj)

User N - Request N

Controller

Constructor(ITransientObject obj, IOtherObject otherObj1, IOtherObject otherObj2,...)

- If the constructor have the transient object injected, it will be creating and will be used for each method in the controller.



Transient Object

- If the constructor (always taking in constructor injection but can be method or property injection) have other object that in their constructors have injected the transient object, then each object will create an instance of transient object, so can exists many instances of transient object in the same request.

otherObj1



Transient Object

otherObj2



Transient Object

Share Improve this answer Follow

answered Apr 22, 2023 at 10:59



Juan Carlos Marciano
Trias

149 ● 2 ● 2



Probably the best illustration of the lifetime comes into play with the EntityFramework/Core via DbContext.

2



It is recommended that DbContext and repositories that interact with DbContext should be wired up with a Scoped lifetime because a DbContext is obviously a stateful construct. So you'd not want to use a Singleton because you'd end up with all kinds of concurrency issues. You'd not want to use Transient because DbContext is not thread safe. Remember, Transient is for use cases where you're dealing with stateless objects/classes.



And since most repositories are called by controllers, it really makes sense to use a Scoped lifetime. It's conceivable that a DbContext could be invoked multiple times during a single action method as part of a transaction.

This article doesn't speak directly about these lifetimes but gives a great explanation for why a Scoped lifetime is most appropriate for DbContext.

<https://mehdi.me/ambient-dbcontext-in-ef6/?msclkid=00251b05d01411ec8d85d232374f26d5>

Share Improve this answer Follow

answered May 10, 2022 at 4:03



Charles Owen

2,792 ● 2 ● 22 ● 29



Highly active question. Earn 10 reputation (not counting the [association bonus](#)) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.