Understanding AddTransient Vs AddScoped Vs AddSingleton In ASP.NET Core

Anupam Maiti Sep 24, 2024 536.4k 18 37

Introduction

Understanding the life cycle of Dependency Injection (DI) is very important in ASP.Net Core applications. As we know, Dependency injection (DI) is a technique for achieving loose coupling between objects and their collaborators, or dependencies. Most often, classes will declare their dependencies via their constructor, allowing them to follow the Explicit Dependencies Principle. This approach is known as "constructor injection".

To implement dependency injection, we need to configure a DI container with classes that is participating in DI. DI Container has to decide whether to return a new instance of the service or provide an existing instance. In startup class, we perform this activity on ConfigureServices method.

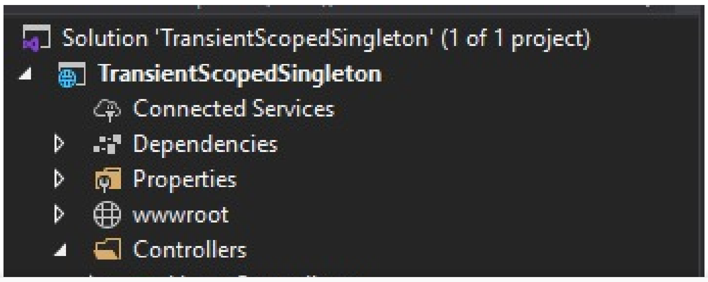
The lifetime of the service depends on when the dependency is instantiated and how long it lives. And lifetime depends on how we have registered those services.

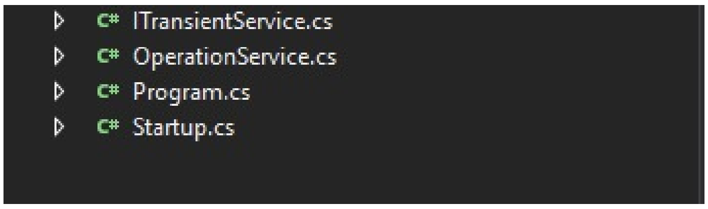
The below three methods define the lifetime of the services,

1. **AddTransient:** Transient lifetime services are created each time they are requested. This lifetime works best for lightweight, stateless services.
2. **AddScoped:** Scoped lifetime services are created once per request.
3. **AddSingleton:** Singleton lifetime services are created the first time they are requested (or when ConfigureServices is run if you specify an instance there) and then every subsequent request will use the same instance.

Understanding Dependency Injection Lifetime with an Example

Let’s understand DI lifetime with an example in a sample ASP.NET core application.





In my sample project, I added three interfaces called – ItransientService, IScopedService, ISingletonService, which represents one of each of the DI lifetimes. Those interfaces contain a single method called GetOperationID(); which will return a uniqueGuid.

```
using System;
namespace TransientScopedSingleton {
    publicinterfaceITransientService {
        Guid GetOperationID();
    }
}
```

```
using System;
namespace TransientScopedSingleton {
    publicinterfaceIScopedService {
        Guid GetOperationID();
    }
}
```

```
using System;
namespace TransientScopedSingleton {
    publicinterfaceISingletonService {
        Guid GetOperationID();
    }
}
```

Let’s implement those 3 interfaces into a service called OperationService.

```
using System;
namespace TransientScopedSingleton {
    publicclassOperationService: ITransientService,
    IScopedService,
    ISingletonService {
        Guid id;
        publicOperationService() {
            id = Guid.NewGuid();
        }
        public Guid GetOperationID() {
            return id;
        }
    }
}
```

Now, register the OperationService via those three interfaces as shown below in the ConfigureServices method of the startup class.

```
services.AddTransient<ITransientService, OperationService>();
services.AddScoped<IScopedService, OperationService>();
services.AddSingleton<ISingletonService, OperationService>();
```

Awesome! Now we are ready to inject those services into controller. For better understanding, we will inject two instances of each service in the constructor of HomeController.

```
private readonly ILogger<HomeController> _logger;
private readonly ITransientService _transientService1;
private readonly ITransientService _transientService2;
private readonly IScopedService _scopedService1;
private readonly IScopedService _scopedService2;
private readonly ISingletonService _singletonService1;
private readonly ISingletonService _singletonService2;

public HomeController(ILogger<HomeController> logger,
            ITransientService transientService1,
            ITransientService transientService2,
            IScopedService scopedService1,
            IScopedService scopedService2,
            ISingletonService singletonService1,
            ISingletonService singletonService2)
{
    _logger = logger;
    _transientService1 = transientService1;
    _transientService2 = transientService2;
    _scopedService1 = scopedService1;
    _scopedService2 = scopedService2;
    _singletonService1 = singletonService1;
    _singletonService2 = singletonService2;
}
```

Now we will call GetOperationID method of each service instance and assign it to viewbag so that we can see those values in UI.

```
public IActionResult Index()
{
    ViewBag.transient1 = _transientService1.GetOperationID().ToString();
    ViewBag.transient2 = _transientService2.GetOperationID().ToString();
    ViewBag.scoped1 = _scopedService1.GetOperationID().ToString();
    ViewBag.scoped2 =  _scopedService2.GetOperationID().ToString();
    ViewBag.singleton1 = _singletonService1.GetOperationID().ToString();
    ViewBag.singleton2 = _singletonService2.GetOperationID().ToString();
    return View();
}
```

Modify view to display ids for respective service type.

```
<div class="text-center">
    <h2 class="display-4">Dependency Injection Lifetime
    </h2>
</div>
<table class="table table-bordered">
    <thead>
        <tr>
            <th>Service Type</th>
            <th>First Instance Operation ID</th>
            <th>Second Instance Operation ID</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <tdstyle="background-color: darksalmon">Transient
            </td>
            <tdstyle="background-color: darksalmon">@ViewBag.transient1
            </td>
            <tdstyle="background-color: darksalmon">@ViewBag.transient2
            </td>
        </tr>
        <tr>
            <td>Scoped</td>
            <td>@ViewBag.scoped1</td>
            <td>@ViewBag.scoped2</td>
        </tr>
        <tr>
            <tdstyle="background-color: aquamarine">Singleton
            </td>
            <tdstyle="background-color: aquamarine">@ViewBag.singleton1
            </td>
            <tdstyle="background-color: aquamarine">@ViewBag.singleton2
            </td>
        </tr>
    </tbody>
</table>
```

Once we execute the application, we will see two different Guids are displayed for their respective service types. Now run two instance of UI in two different tabs of the browser like request 1 and request 2.

Request 1

Dependency Injection in ASP.NET Core,AddTransient,AddScoped,AddSingleton,Dependency Injection Lifetime

Request 2

Dependency Injection in ASP.NET Core,AddTransient,AddScoped,AddSingleton,Dependency Injection Lifetime

Observation from Request 1 and Request 2

Transient service always returns a new instance even though it’s the same request, that is why operation Ids are different for first instance and second instance for both the requests (Request 1 and Request 2).

In the case of Scoped service, a single instance is created per request and the same instance is shared across the request. That is why operation Ids are the same for first instance as well as second instance of Request 1. But if we click on refresh button or load the UI on different tab of a browser (which is nothing but Request 2), new ids are generated.

In the case of Singleton service, only one instance is created and shared across applications. If we click on refresh button or load the UI on a different tab of a browser (which is nothing but Request 2), those ids will remain the same.

Service Type	In the scope of a given HTTP request	Across different HTTP requests
Transient	New Instance	New Instance
Scoped	Same Instance	New Instance
Singleton	Same Instance	Same Instance

When to use what?

Transient Scenarios

- **Formatting Operations**: Services that format strings, dates, or other data types.
- **Helper Services**: Services that provide utility methods, such as generating random numbers or unique IDs.

Scoped Scenarios

- **Database Context**: Using Entity Framework Core's `DbContext` to manage database operations. Each request gets its own context to ensure changes are isolated to that request.
- **Unit of Work Pattern**: Managing a set of operations that must be completed in a single transaction.

Singleton Scenarios

- **Configuration Service**: Managing application-wide settings and configuration that do not change during the application's lifetime.
- **Caching Service**: Storing and retrieving data that needs to be shared across the application.

Summary

Let's summarize what we discussed so far,

- With a transient service, a new instance is provided every time an instance is requested whether it is in the scope of same HTTP request or across different HTTP requests.
- With a scoped service we get the same instance within the scope of a given HTTP request but a new instance across different HTTP requests.
- With Singleton service, there is only a single instance. An instance is created, when service is first requested and that single instance single instance will be used by all subsequent HTTP request throughout the application.

Happy Reading!

AddScoped AddSingleton AddTransient Dependency Injection in ASP.NET Core
Dependency Injection Lifetime

SIMILAR ARTICLES

AddTransient vs AddScoped vs AddSingleton

AddTransient and AddScoped for Repository Registration in ASP.NET Core

AddSingleton Vs AddScoped Vs AddTransient

Dependency Injection Lifetimes In ASP.NET CORE

ASP.NET Or ASP.NET Core, What To Choose?

Anupam Maiti TOP 500

Anupam is a technology enthusiast with a positive attitude, constantly seeking growth and learning opportunities. His expertise lies in successfully executing challenging Cloud Assessment & Migration, Cloud Native En... Read more

182 3.4m 4

18 View All Comments

Type your comment here and press Enter Key (Minimum 10 characters)