ASP.NET Core Dependency Injection Best Practices, Tips & Tricks

Halil İbrahim Kalkan in Volosoft Follow
Jul 12, 2018 · 7 min read

In this article, I will share my experiences and suggestions on using Dependency Injection in ASP.NET Core applications. The motivation behind these principles are;

- Effectively designing services and their dependencies.
- Preventing multi-threading issues.
- Preventing memory-leaks.
- Preventing potential bugs.

This article assumes that you are already familiar with Dependency Injection and ASP.NET Core in a basic level. If not, please read the ASP.NET Core Dependency Injection documentation first.

Basics

Constructor Injection

Constructor injection is used to declare and obtain dependencies of a service on the **service construction.** Example:

```
public class ProductService
{
    private readonly IProductRepository _productRepository;
    public ProductService(IProductRepository productRepository)
    {
        _productRepository = productRepository;
    }
    public void Delete(int id)
    {
        _productRepository.Delete(id);
    }
}
```

- Define **required dependencies** explicitly in the service constructor. Thus, the service can not be constructed without its dependencies.
- Assign injected dependency to a **read only** field/property (to prevent accidentally assigning another value to it inside a method).

Property Injection

ASP.NET Core's standard dependency injection container **does not support property injection**. But you can use another container supporting the property injection. Example:

```
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Logging.Abstractions;
namespace MyApp
{
    public class ProductService
    {
        public ILogger<ProductService> Logger { get; set; }
        private readonly IProductRepository _productRepository;
        public ProductService(IProductRepository productRepository)
        {
            _productRepository = productRepository;
            Logger = NullLogger<ProductService>.Instance;
        }
        public void Delete(int id)
        {
            _productRepository.Delete(id);
            Logger.LogInformation(
                $"Deleted a product with id = {id}");
        }
    }
}
```

ProductService is declaring a Logger property with **public setter.** Dependency injection container can set the Logger if it is available (registered to DI container before).

Good Practices:

- Use property injection **only for optional dependencies.** That means your service can properly work **without** these dependencies provided.
- Use Null Object Pattern (as like in this example) if possible. Otherwise, always check for null while using the dependency.

Service Locator

Service locator pattern is another way of obtaining dependencies. Example:

```
public class ProductService
{
    private readonly IProductRepository _productRepository;
    private readonly ILogger<ProductService> _logger;
    public ProductService(IServiceProvider serviceProvider)
    {
        _productRepository = serviceProvider
          .GetRequiredService<IProductRepository>();
        _logger = serviceProvider
          .GetService<ILogger<ProductService>>() ??
            NullLogger<ProductService>.Instance;
    }
```

```
            _logger.LogInformation($"Deleted a product with id = {id}");
        }
    }
```

ProductService is injecting **IServiceProvider** and resolving dependencies using it.

GetRequiredService throws exception if the requested dependency was not registered before. On the other hand, **GetService** just returns null in that case.

When you resolve services inside the **constructor**, they are released when the service is released. So, you don't care about releasing/disposing services resolved inside the constructor (just like constructor and property injection).

Good Practices:

- **Do not use** the service locator pattern wherever possible (if the service type is known in the development time). Because it makes the dependencies **implicit**. That means it's not possible to see the dependencies easily while creating an instance of the service. This is especially important for **unit tests** where you may want to **mock** some dependencies of a service.
- Resolve dependencies in the service **constructor** if possible. Resolving in a **service method** makes your application more complicated and error prone. I will cover the problems & solutions in the next sections.

Service Life Times

There are three service lifetimes in ASP.NET Core Dependency Injection:

1. **Transient** services are created every time they are injected or requested.
2. **Scoped** services are created per scope. In a web application, every web request creates a new separated service scope. That means scoped services are generally created per web request.
3. **Singleton** services are created per DI container. That generally means that they are created only one time per application and then used for whole the application life time.

DI container keeps track of all resolved services. Services are released and disposed when their lifetime ends:

- If the service has **dependencies**, they are also automatically released and disposed.
- If the service implements the **IDisposable** interface, Dispose method is automatically called on service release.

Good Practices:

- Register your services as **transient** wherever possible. Because it's simple to design transient services. You generally don't care about **multi-threading** and **memory leaks** and you know the service has a short life.
- Use **scoped** service lifetime **carefully** since it can be tricky if you create child service scopes or use these services from a non-web application.
- Use **singleton** lifetime carefully since then you need to **deal** with **multi-threading** and potential **memory leak** problems.
- **Do not depend** on a transient or scoped service from a singleton service. Because the transient service becomes a singleton instance when a singleton service injects it and that may cause problems if the transient service is not designed to support such a scenario. ASP.NET Core's default DI container already throws **exceptions** in such cases.

Resolving Services in a Method Body

In some cases, you may need to resolve another service in a method of your service. In such cases, ensure that you release the service after usage. The best way of ensuring that is to create a **service scope**. Example:

```
    public PriceCalculator(IServiceProvider serviceProvider)
    {
        _serviceProvider = serviceProvider;
    }
    public float Calculate(Product product, int count,
      Type taxStrategyServiceType)
    {
        using (var scope = _serviceProvider.CreateScope())
        {
            var taxStrategy = (ITaxStrategy)scope.ServiceProvider
              .GetRequiredService(taxStrategyServiceType);
            var price = product.Price * count;
            return price + taxStrategy.CalculateTax(price);
        }
    }
}
```

PriceCalculator injects the **IServiceProvider** in its constructor and assigns it to a field. PriceCalculator then uses it inside the Calculate method to create a **child service scope**. It uses **scope.ServiceProvider** to resolve services, instead of the injected _serviceProvider instance. Thus, all services resolved from the scope is automatically released/disposed at the end of the **using** statement.

Good Practices:

- If you are resolving a service in a method body, always create a **child service scope** to ensure that the resolved services are properly released.
- If a method gets **IServiceProvider** as an argument, then you can directly resolve services from it without care about releasing/disposing. Creating/managing service scope is a responsibility of the code calling your method. Following this principle makes your code cleaner.
- **Do not hold a reference to a resolved service!** Otherwise, it may cause memory leaks and you will access to a **disposed service** when you use the object reference later (unless the resolved service is singleton).

Singleton Services

Singleton services are generally designed to keep an application state. A cache is a good example of application states. Example:

```
public class FileService
{
    private readonly ConcurrentDictionary<string, byte[]> _cache;
    public FileService()
    {
        _cache = new ConcurrentDictionary<string, byte[]>();
    }
    public byte[] GetFileContent(string filePath)
    {
        return _cache.GetOrAdd(filePath, _ =>
        {
            return File.ReadAllBytes(filePath);
        });
    }
}
```

Good Practices:

- If the service holds a state, it should access to that state in a **thread-safe** manner. Because all requests concurrently uses the **same instance** of the service. I used **ConcurrentDictionary** instead of Dictionary to ensure thread safety.
- **Do not use scoped or transient services** from singleton services. Because, transient services might not be designed to be thread safe. If you have to use them, then take care of multi-threading while using these services (use lock for instance).
- **Memory leaks** are generally caused by singleton services. They are not released/disposed until the **end of the application**. So, if they instantiate classes (or inject) but not release/dispose them, they will also stay in the memory until the end of the application. Ensure that you **release/dispose** them at the right time. See the *Resolving Services in a Method Body* section above.
- If you cache data (file contents in this example), you should create a mechanism to update/invalidate the cached data when the original data source changes (when a cached file changes on the disk for this example).

Scoped Services

Scoped lifetime **first seems** a good candidate to store per web request data. Because ASP.NET Core creates a **service scope per web request**. So, if you register a service as scoped, it can be shared during a web request. Example:

```
public class RequestItemsService
{
    private readonly Dictionary<string, object> _items;
    public RequestItemsService()
    {
        _items = new Dictionary<string, object>();
    }
    public void Set(string name, object value)
    {
        _items[name] = value;
    }
    public object Get(string name)
    {
        return _items[name];
    }
}
```

If you register the RequestItemsService as scoped and inject it into two different services, then you can get an item that is added from another service because they will share the same RequestItemsService instance. That's what we expect from scoped services.

But.. the fact may not be always like that. If you create a **child service scope** and resolve the RequestItemsService from the child scope, then you will get a new instance of the RequestItemsService and it will not work as you expect. So, scoped service does not always means instance per web request.

You may think that you do not make such an obvious mistake (resolving a scoped inside a child scope). But, this is not a mistake (a very regular usage) and the case may not be such simple. If there is a big dependency graph between your services, you can not know if anybody created a