An Empirical Study of Dependency Injection Lifetime Effects in C# Applications

AzraJabeen Mohamed Ali

Independent researcher, California, USA
Azra.jbn@gmail.com

Abstract

This empirical study investigates the effects of different Dependency Injection (DI) lifetimes in C# applications, focusing on their impact on performance, resource management, and maintainability. DI lifetimes—such as Transient, Scoped, and Singleton—play a crucial role in the lifecycle of objects and services within a system. While DI frameworks, such as Microsoft. Extensions. Dependency Injection, offer flexibility in managing these lifetimes, the practical consequences of each scope on large-scale C# applications remain underexplored. Through a series of controlled experiments, we analyze key metrics including memory usage, application startup time, response times, and ease of unit testing across varying lifetimes. The study also examines the trade-offs between performance and maintainability, providing insights into best practices for choosing appropriate lifetimes in different architectural contexts. Our findings reveal how improper DI lifetime choices can lead to resource inefficiencies and reduced application scalability, while also highlighting the benefits of well-managed scopes for improving code quality and testability. Ultimately, this research contributes to a deeper understanding of DI lifetime management and offers actionable recommendations for developers to optimize their C# applications' design and performance.

Keywords: Dependency Injection, lifetime, Transient, Scoped, Singleton, lifecycle, Service provider, Service Collection

1. Introduction

Dependency Injection (DI) is a core design pattern widely used in modern software development, especially in object-oriented and component-based systems. It facilitates loose coupling, increases modularity, and promotes better testability by allowing the injection of dependencies rather than hard-coding them within classes. In C# applications, DI has become an essential feature, largely through frameworks like Microsoft.Extensions.DependencyInjection, which provides a flexible and powerful system for managing object lifetimes.

In DI, the concept of *lifetime* determines how and when objects are created and managed by the container, affecting their scope and lifecycle within the application. The three primary lifetimes available in DI frameworks—Transient, Scoped, and Singleton—each offer distinct trade-offs in terms of object reuse, memory management, and performance. A **Transient** service is created each time it is requested, while a **Scoped** service is created once per request or operation scope. A **Singleton** service is created once and reused throughout the application's lifetime.

Despite the extensive use of DI in C# applications, there is limited empirical research that evaluates the real-world consequences of these lifetimes in terms of performance, memory consumption, and code maintainability. While it is well-understood that improper DI lifetime management can lead to issues such as memory leaks or poor performance, concrete, data-driven evidence is sparse. This gap in knowledge is particularly critical in large-scale applications where even small inefficiencies can accumulate, impacting overall system performance.
This study aims to fill this gap by conducting an empirical investigation into the effects of different DI lifetimes in C# applications. Through a series of controlled experiments, we examine how each lifetime influences key performance indicators, including memory usage, initialization time, request handling time, and the complexity of unit testing. Additionally, we explore the trade-offs involved in choosing between lifetimes, highlighting the broader implications of DI lifetime decisions for maintainable and scalable software architecture.
By providing data-driven insights into the impact of DI lifetimes, this research will help developers and architects make informed decisions when configuring dependency injection in their C# applications, leading to more efficient and sustainable software systems.

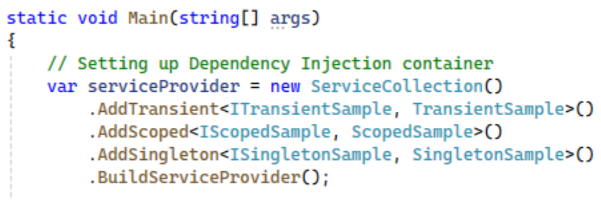
Dependency Injection:
In simple terms, an object that is dependent on another is called a dependence.
Microsoft.Extensions.DependencyInjection is a package in the .NET ecosystem that provides a built-in, lightweight dependency injection (DI) framework. It is commonly used in ASP.NET Core applications but can be utilized in any .NET application that requires dependency injection. It is necessary to make sure to have the Microsoft.Extensions.DependencyInjection package installed. In .NET Core, this package is included by default in ASP.NET Core, but for a console application, it is to be installed in it.

Dependency Injection Container:
The Dependency Injection (DI) Container is a core part of the Dependency Injection pattern in .NET, provided by the Microsoft.Extensions.DependencyInjection package. The DI container is responsible for managing the lifecycle of dependencies in an application, resolving instances of services, and injecting them where needed.

How the Dependency Injection Container Works:

1. **Service Registration:** Services are registered in the container with their interfaces and implementations. It is possible to define the service lifetime (Transient, Scoped, Singleton). ServiceCollection is a class that holds service registrations. It is used to add services to the container. It provides methods to register services and specify their lifetime. After configuring the services, the container can be built using BuildServiceProvider() as per Fig -1. The ServiceProvider is the container that holds the registered services and is responsible for resolving and providing instances of services when they are requested. It is created by calling .BuildServiceProvider() on the ServiceCollection.

Fig-1:



```
static void Main(string[] args)
{
    // Setting up Dependency Injection container
    var serviceProvider = new ServiceCollection()
        .AddTransient<ITransientSample, TransientSample>()
        .AddScoped<IScopedSample, ScopedSample>()
        .AddSingleton<ISingletonSample, SingletonSample>()
        .BuildServiceProvider();
```

2. **Service Resolution:** Once the services are registered, the DI container can resolve and provide these services when requested. This is done through constructor injection, method injection, or property injection.

Service Lifetimes in Dependency Injection Container:

The container allows to define how long instances of services should live. There are three main service lifetimes in .NET:

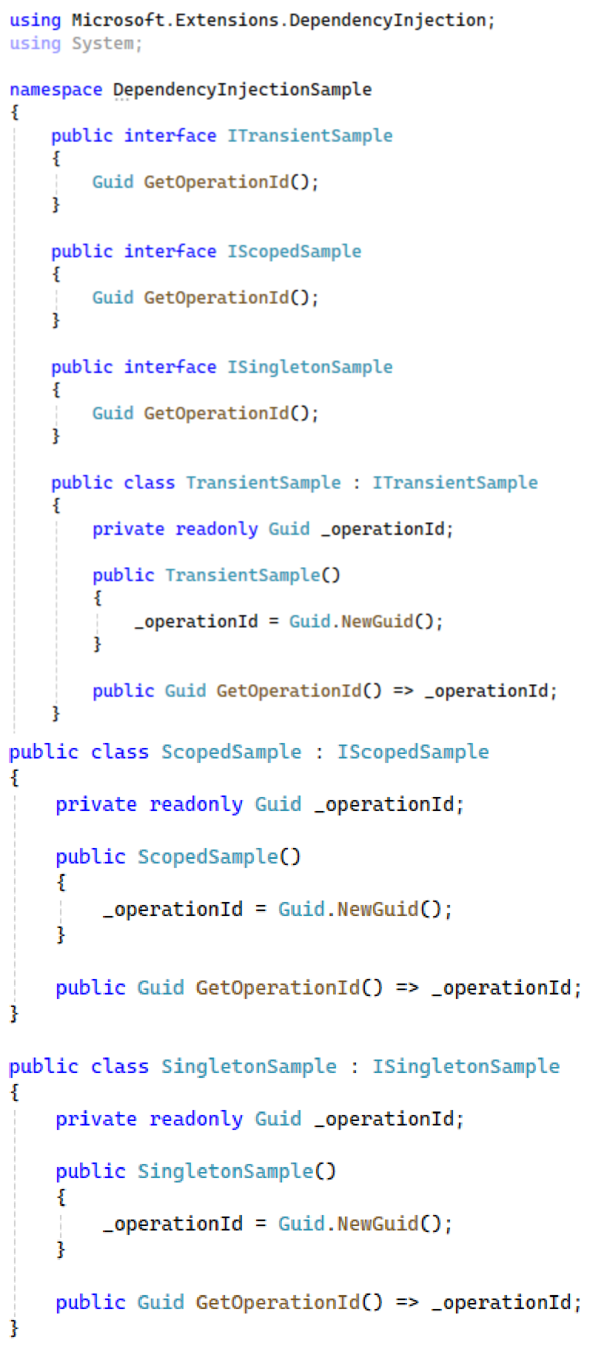
1. **Transient**: A new instance of the service is created each time it is requested.
2. **Scoped**: A new instance is created once per scope (often used for per-request lifetimes in web applications).
3. **Singleton**: A single instance is created and shared throughout the application’s lifetime.

Code Sample:

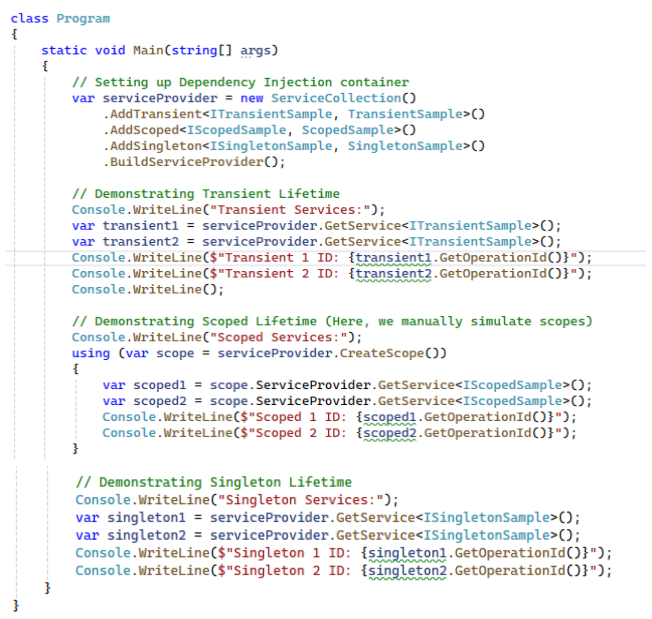
Demonstrating the usage of dependency injection with different service lifetimes as per Fig-2.

1. **Service Interfaces**: The ITransientSample, IScopedSample, and ISingletonSample interfaces define the methods that will be implemented by the services.
2. **Service Implementations**:
- **TransientSample**: Implements ITransientSample. A new instance is created each time.
- **ScopedSample**: Implements IScopedSample. A new instance is created per scope (typically per HTTP request in web apps).
- **SingletonSample**: Implements ISingletonSample. A single instance is shared throughout the application.
3. **Service Registration**: In the Main method, services are registered with different lifetimes using AddTransient, AddScoped, and AddSingleton.
4. **Service Resolution**:
- Transient services are created anew each time they are requested.
- Scoped services are shared within the same scope but different across scopes.
- Singleton services are created once and shared throughout the application's lifetime.

Fig-2:



```
using Microsoft.Extensions.DependencyInjection;
using System;

namespace DependencyInjectionSample
{
    public interface ITransientSample
    {
        Guid GetOperationId();
    }

    public interface IScopedSample
    {
        Guid GetOperationId();
    }

    public interface ISingletonSample
    {
        Guid GetOperationId();
    }

    public class TransientSample : ITransientSample
    {
        private readonly Guid _operationId;

        public TransientSample()
        {
            _operationId = Guid.NewGuid();
        }

        public Guid GetOperationId() => _operationId;
    }
public class ScopedSample : IScopedSample
{
    private readonly Guid _operationId;

    public ScopedSample()
    {
        _operationId = Guid.NewGuid();
    }

    public Guid GetOperationId() => _operationId;
}

public class SingletonSample : ISingletonSample
{
    private readonly Guid _operationId;

    public SingletonSample()
    {
        _operationId = Guid.NewGuid();
    }

    public Guid GetOperationId() => _operationId;
}
```

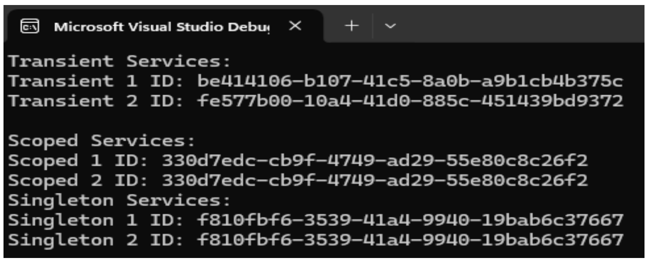


```
class Program
{
    static void Main(string[] args)
    {
        // Setting up Dependency Injection container
        var serviceProvider = new ServiceCollection()
            .AddTransient<ITransientSample, TransientSample>()
            .AddScoped<IScopedSample, ScopedSample>()
            .AddSingleton<ISingletonSample, SingletonSample>()
            .BuildServiceProvider();

        // Demonstrating Transient Lifetime
        Console.WriteLine("Transient Services:");
        var transient1 = serviceProvider.GetService<ITransientSample>();
        var transient2 = serviceProvider.GetService<ITransientSample>();
        Console.WriteLine($"Transient 1 ID: {transient1.GetOperationId()}");
        Console.WriteLine($"Transient 2 ID: {transient2.GetOperationId()}");
        Console.WriteLine();

        // Demonstrating Scoped Lifetime (Here, we manually simulate scopes)
        Console.WriteLine("Scoped Services:");
        using (var scope = serviceProvider.CreateScope())
        {
            var scoped1 = scope.ServiceProvider.GetService<IScopedSample>();
            var scoped2 = scope.ServiceProvider.GetService<IScopedSample>();
            Console.WriteLine($"Scoped 1 ID: {scoped1.GetOperationId()}");
            Console.WriteLine($"Scoped 2 ID: {scoped2.GetOperationId()}");
        }

        // Demonstrating Singleton Lifetime
        Console.WriteLine("Singleton Services:");
        var singleton1 = serviceProvider.GetService<ISingletonSample>();
        var singleton2 = serviceProvider.GetService<ISingletonSample>();
        Console.WriteLine($"Singleton 1 ID: {singleton1.GetOperationId()}");
        Console.WriteLine($"Singleton 2 ID: {singleton2.GetOperationId()}");
    }
}
```

Output:

As per output in Fig-3, Transient services produce a new instance each time they are requested. Scoped services share an instance within the same scope but are different across different scopes. Singleton services share a single instance for the entire application.

Fig-3:



```
Transient Services:
Transient 1 ID: be414106-b107-41c5-8a0b-a9b1cb4b375c
Transient 2 ID: fe577b00-10a4-41d0-885c-451439bd9372

Scoped Services:
Scoped 1 ID: 330d7edc-cb9f-4749-ad29-55e80c8c26f2
Scoped 2 ID: 330d7edc-cb9f-4749-ad29-55e80c8c26f2
Singleton Services:
Singleton 1 ID: f810fbf6-3539-41a4-9940-19bab6c37667
Singleton 2 ID: f810fbf6-3539-41a4-9940-19bab6c37667
```

When to Choose Transient Service:

- **Stateless Services:** If a service does not maintain any internal state and doesn't depend on long-lived objects or shared resources.
- **Short-Lived Operations:** Services that are used temporarily for a single operation or request and can be discarded after the operation is complete.
- **Independence:** If you want each consumer of the service to have a separate instance, ensuring no shared state or side effects between consumers.

Examples of Transient Service:

- Logging (if it does not store state between logs).
- Simple utility classes (e.g., a service that formats strings or performs calculations).
- Services that do not hold or require expensive resources or complex state management.

When to choose Scoped Service:

- **Per-Request or Operation Context:** Services that should be created once per operation (e.g., an HTTP request or background job), but reused within the context of that operation.
- **Shared State Within Scope**: When you need to maintain state between multiple components or services for the duration of the scope (e.g., a database transaction that should be consistent within one request).
- **Unit of Work**: Scoped services are ideal for situations that require consistency of data across multiple steps in a single transaction or request, like managing a database context (e.g., Entity Framework Core's DbContext).

Examples of Scoped Service:

- An Entity Framework DbContext is often scoped, as it needs to persist data and manage transactions within a specific request.
- The user's authentication information should be consistent during a request but different across requests.
- A service that coordinates multiple repositories or actions in a single request.

When to choose Singleton Service:

- **Global Shared State:** Services that maintain state that is shared across all components in the application and should be initialized only once.
- **Heavy Resource Management:** If the service involves expensive initialization or heavy resources (e.g., database connections, external API connections), creating it once and sharing the instance throughout the application's lifetime can improve performance.
- **Thread-Safe and Immutable:** The service must be either stateless or be thread-safe if it manages mutable state because it will be used by multiple threads concurrently.

Examples of Singleton Service:

- Services that read from configuration files or environment variables and provide configuration values throughout the app.
- A centralized logging service, where a single instance should handle logging messages across the entire application.
- A caching service that stores data for later retrieval and is used throughout the application.

Benefits of Dependency Injection (DI):

- **Loose Coupling:** Dependency Injection helps achieve loose coupling between classes, meaning that classes do not directly create their dependencies. Instead, dependencies are injected from the outside, reducing the interdependencies between components. This makes the system more flexible and easier to maintain because changes in one component are less likely to affect others.

- **Improved Testability:** One of the key advantages of DI is its support for unit testing. Since dependencies can be injected at runtime, it becomes easy to mock or stub these dependencies in tests. This results in more isolated and controlled tests, allowing developers to verify the behavior of individual components without worrying about their dependencies.

- **Increased Maintainability:** By decoupling components and promoting modular design, DI leads to better code organization. When the responsibilities of a class are clearly separated and dependencies are injected rather than hard-coded, it becomes easier to understand and maintain the codebase. Changes to the implementation of dependencies can be made independently, without affecting the dependent classes.

- **Reusability:** DI promotes the creation of reusable and interchangeable components. Since dependencies are passed in rather than created within the class, different implementations of an interface or service can be injected depending on the context. This allows for greater flexibility in reusing components across different parts of the application.

- **Centralized Configuration:** With DI, the management and configuration of dependencies can be centralized. Rather than configuring dependencies manually in each class, the DI container or framework (such as Microsoft.Extensions.DependencyInjection) can manage the instantiation and lifecycle of objects. This centralization simplifies configuration management and makes it easier to change how dependencies are resolved across the application.

- **Separation of Concerns:** DI helps achieve a clear separation of concerns by isolating the logic of object creation from the logic of the application. Classes focus on their core responsibilities and are not concerned with how their dependencies are created, allowing for cleaner and more maintainable code.

- **Easier Refactoring:** DI enables more straightforward refactoring since dependencies are injected and not tightly coupled with the class implementation. This allows developers to replace or modify components with minimal changes to the classes that rely on them, simplifying the process of refactoring and improving long-term code quality.

- **Reduced Code Duplication:** By relying on a DI framework to inject shared dependencies, developers avoid redundant code in constructors and across classes. This reduces code duplication and promotes a cleaner, more DRY (Don't Repeat Yourself) codebase.

- **Better Resource Management (with Dependency Lifetimes):** DI frameworks allow for control over the lifetimes of dependencies (e.g., Transient, Scoped, Singleton). This enables efficient resource management by ensuring that objects are created, reused, or disposed of at the appropriate times, which can lead to improved performance and reduced memory overhead.

Conclusion:

The empirical analysis suggests that the choice of Dependency Injection service lifetime should be carefully considered based on the specific requirements of the application. Service lifetimes should be selected to strike a balance between performance, resource management, and maintainability. By understanding the effects of different lifetimes, developers can make informed decisions to optimize their applications, ensuring scalability and robustness in real-world scenarios. Selecting the correct DI lifetime is essential for optimizing performance, resource management, and maintainability. By understanding the strengths and trade-offs of Transient, Scoped, and Singleton lifetimes, developers can create more efficient, scalable, and maintainable applications. The correct use of DI can simplify the development process, improve testing, and ensure that services are managed in the most effective way.

References

[1] Elisenda Gascon, "Service Lifetimes in ASP.NET Core" https://endjin.com/blog/2022/09/service-lifetimes-in-aspnet-core (Sep 06, 2022)

[2] Henrique Siebert Domareski, "Dependency Injection and Service Lifetimes in .NET Core" https://henriquesd.medium.com/dependency-injection-and-service-lifetimes-in-net-core-ab9189349420 (Nov 25, 2020)

[3] Davide Bellone "Dependency Injection lifetimes in .NET - Scoped vs Transient vs Singleton" https://www.code4it.dev/blog/dependency-injection-lifetimes/#google_vignette (May 26, 2020)

[4] Derek Comartin "Thread Safety & Scoped Lifetime in Dependency Injection Containers" https://codeopinion.com/thread-safety-scoped-lifetime-in-dependency-injection-containers/ (Jul 15, 2020)

[5] Microsoft "ASP.NET Core Pitfalls – Dependency Injection Lifetime Validation" https://weblogs.asp.net/ricardoperes/asp-net-core-pitfalls-dependency-injection-lifetime-validation (Nov 24, 2020)

[6] Ivan Stove "Dependency Injection Lifetimes in .NET" https://levelup.gitconnected.com/cdependency-injection-lifetimes-in-net-566830a633ca (May 24, 2022)

[7] Michał Dudak "Dependency lifetime in ASP.NET Core" https://blog.dudak.me/2018/dependency-lifetime-in-asp-net-core/ (Feb 18, 2018)

[8] Madhavan Nagarajan ".Net core Dependency Injection: Lifetimes and few best practices" https://levelup.gitconnected.com/net-core-dependency-injection-lifetimes-and-few-best-practices-e194d2a39eff (Nov 1, 2020)

[9] Andrea Chiarelli "Understanding Dependency Injection in .NET Core" https://auth0.com/blog/dependency-injection-in-dotnet-core/ (Feb 11, 2020)

[10] Ervis Trupja "Understanding the Differences between Singleton, Scoped, and Transient Service Lifetime in .NET" https://dotnethow.net/understanding-the-differences-between-singleton-scoped-and-transient-service-lifetime-in-net/ (Jan 18, 2023)

[11] Nirjhar Choudhury "Service Lifetime in Dependency Injection " https://dotnetcorecentral.com/blog/service-lifetime/#google_vignette (Jan 30, 2021)

[12] Mark Seemann, Steven van Deursen, "Dependency Injection Principles, Practices, and Patterns" Manning Publications (Mar 16, 2019)

[13] Mark Seemann, "Dependency Injection in .NET 1st Edition" Manning Publications (Sep 28, 2011)

[14] Marino Posadas, Tadit Dash "Dependency Injection in .NET Core 2.0: Make use of constructors, parameters, setters, and interface injection to write reusable and loosely-coupled code" Packt Publications (Nov 13, 2017)

[15] Robert Martin "Clean Architecture: A Craftsman's Guide to Software Structure and Design (Robert C. Martin Series) 1st Edition" Pearson publisher (Sep 10, 2017)