Dependency Inversion and Dependency Injection

Dependency injection is a concept in Object Oriented Programming (OOP) used as a design pattern to reduce the coupling between components. The main goal of dependency injection is to decouple class dependencies from the class itself and achieve inversion of control.

Inversion of control (IoC)

The principle of IoC is to invert the flow of control in the program by providing the dependencies to the class externally instead of the class instantiating them directly.

Dependency

An object or a service that is essential for the class to function.

Injection

Using different methods to provide a dependency to a class, these methods include constructor injection, property injection, and method injection.

Service container (IoC Container)

Manages the life cycle of objects, resolves dependencies and inject them to the classes that need them.

Types of Dependency Injection

1. **Constructor injection:** the dependencies are provided to the class through its constructor.

```
Class Car
{
      Private readonly IEngine _engine;
      Public Car (IEngine engine)
      {
            _engine = engine;
```

```
        }
}
```

2. **Property Injection:** making dependencies as public properties to be set externally.

```
Class Car
{
        public IEngine _engine { get; set; };
}
```

3. **Method Injection:** instead of using a constructor, dependencies can be passed through methods. Useful for when the dependency is required for a specific method only and not the entire object.

```
Class Car
{
        public void SetEngine (IEngine engine)
        {
                _engine = engine;
        }
}
```

Dependency injection can be done using IoC containers to register and configure the services. The dependency life cycle is decided in this part.

```
Class Program
{
    Public void ConfigureServices (IServiceCollection
services)
    {
        services.AddScoped<IUserService, UserService>( );
        services.AddSingleton<ICar, Car>( );
        services.AddTransient<IEngine, Engine>( );
    }
}
```

Scope and Life Cycle of Services

To optimize and improve the system performance, it is essential to determine the correct lifetime and scope for the services used in the system. There are 3 types service lifetimes:

1. **Scoped:**

   Created once per user request, useful for web applications where the service is needed throughout each request or session to ensure it is tied to a specific context.

2. **Singleton:**

   Created once for all requests and components, used throughout the application's lifetime. Used for applications that need a maintained global state, and shared resource management.

3. **Transient:**

   Created each time it is requested, used for lightweight stateless objects that require short-lived services and do not need persistent data for requests.

Service	Transient	Singleton	Scoped
Lifetime	Created each time it is requested.	Created once and shared throughout application.	Created once per request/operation (e.g., per HTTP request).
State	Stateless.	Maintains global state throughout app lifetime.	Tied to a request or operation, state-specific to that request.
Use Case	Lightweight, stateless services.	Shared resources like caching, logging, or configuration.	Services tied to a user/session (e.g., HTTP request).
Example	Logging, email sender, random number generator.	Configuration manager, logging, caching service.	Order processing, user-specific settings or authentication context

To conclude, dependency Injection is a fundamental pattern in C# and modern .NET development that encourages loose coupling, testability, and maintainability. By using DI, developers can build applications that are more modular and easier to manage over time. The key to effectively using DI is understanding the different ways to inject dependencies, managing the lifetimes of services, and knowing when to apply this pattern to the code.

Choosing the appropriate service lifetime in the application ensures the proper management of resources, minimizes overhead, and avoids problems like memory leaks or inconsistent application behavior. A transient service is best for stateless operations, a singleton for shared, long-lived resources, and a scoped service when the instance needs to be tied to the lifecycle of a specific request or operation.