Service in C#

Service Lifetimes in Dependency Injection

C# supports three main service lifetimes, each defining how long an instance of a service is held in memory:

- **Singleton**: A single instance is created and shared throughout the application's lifetime.
  - **Use case**: Logging, configuration data, or caching data.
  - **Example**:

```
services.AddSingleton<IMyService, MyService>();
```

Scoped: A new instance is created for each HTTP request. This is commonly used in web applications, especially for services that manage database connections or handle request-specific data.

- **Use case**: Database contexts, business logic scoped to requests.
- **Example**:

```
services.AddScoped<IMyService, MyService>();
```

Transient: A new instance is created each time the service is requested. This is suitable for lightweight, stateless services.

- **Use case**: Lightweight, short-lived operations.
- **Example**:

```
services.AddTransient<IMyService, MyService>();
```

2. Common Service Patterns in C#

There are several common patterns used to design services in C# applications, depending on the business logic and data handling requirements.

a. Repository Pattern

The Repository Pattern provides an abstraction over data access logic. It simplifies data access code and decouples the application from specific data sources. Typically, repositories are injected into services, allowing services to focus solely on business logic.

Example:

```
public interface IProductRepository

{

    Product GetProduct(int id);

    void AddProduct(Product product);

    void DeleteProduct(int id);

}
```

```
public class ProductRepository : IProductRepository
{
        private readonly DbContext _context;
        public ProductRepository(DbContext context)
        {
        _context = context;
        }
        public Product GetProduct(int id) => _context.Products.Find(id);
        public void AddProduct(Product product) => _context.Products.Add(product);
        public void DeleteProduct(int id)
        {
        var product = _context.Products.Find(id);
        if (product != null) _context.Products.Remove(product);
        }
}
```

b. Unit of Work Pattern

The Unit of Work pattern groups multiple repository operations within a single transaction, ensuring that all operations succeed or fail together. This pattern is commonly used in complex services that need to perform multiple data operations atomically.

Example:

```
public interface IUnitOfWork : IDisposable

{           IProductRepository ProductRepository { get; }

            void SaveChanges();  }

public class UnitOfWork : IUnitOfWork

{           private readonly DbContext _context;

            private ProductRepository _productRepository;

            public UnitOfWork(DbContext context)

            {           _context = context;    }

            public IProductRepository ProductRepository => _productRepository ??= new ProductRepository(_context);

            public void SaveChanges() => _context.SaveChanges();

            public void Dispose() => _context.Dispose();

}
```

c. Service Layer Pattern

A service layer encapsulates business logic and orchestrates multiple repositories. This layer is used to separate the application's business logic from data access and other cross-cutting concerns.

Example:

```
public interface IOrderService

{

        void PlaceOrder(Order order);

}

public class OrderService : IOrderService

{

        private readonly IUnitOfWork _unitOfWork;

        public OrderService(IUnitOfWork unitOfWork)

        {       _unitOfWork = unitOfWork;     }

        public void PlaceOrder(Order order)

        {       _unitOfWork.ProductRepository.AddProduct(order.Product);

        _unitOfWork.SaveChanges();

        }

}
```