# Dependency Injection

**Dependency Injection (DI)** is a design pattern in which an object receives its dependencies from an external source rather than creating them itself. Dependencies are the objects or services that a class needs to function, and by injecting these dependencies, you decouple the class from the creation process, making the system more modular and flexible.

**How to Achieve Dependency Injection?**

Dependency Injection can be achieved in several ways:

1. **Constructor Injection**:
   * **Description**: The dependencies are provided through a class's constructor. This is the most common method and is often used when the dependency is mandatory for the object to function.
   * **Example**:

csharp

Copy code

public class Car

{

private readonly IEngine \_engine;

public Car(IEngine engine)

{

\_engine = engine;

}

public void Start()

{

\_engine.Run();

}

}

* + In this example, Car depends on an IEngine, which is injected through the constructor.

1. **Property Injection (Setter Injection)**:
   * **Description**: The dependencies are provided through public properties of the class. This is useful when the dependency is optional or can be changed during the object's lifetime.
   * **Example**:

csharp

Copy code

public class Car

{

public IEngine Engine { get; set; }

public void Start()

{

Engine?.Run();

}

}

1. **Method Injection**:
   * **Description**: The dependencies are provided through method parameters. This method is used when the dependency is only required for a specific method.
   * **Example**:

csharp

Copy code

public class Car

{

public void Start(IEngine engine)

{

engine.Run();

}

}

1. **Interface Injection**:
   * **Description**: The dependency provides an injector method that will inject the dependency into any client passed to it. It requires the client to implement an interface that accepts the dependency.
   * **Example**:

csharp

Copy code

public interface IEngineInjectable

{

void InjectEngine(IEngine engine);

}

public class Car : IEngineInjectable

{

private IEngine \_engine;

public void InjectEngine(IEngine engine)

{

\_engine = engine;

}

public void Start()

{

\_engine?.Run();

}

}

**Why Use Dependency Injection?**

Dependency Injection offers several benefits:

1. **Decoupling**:
   * DI decouples the creation of dependencies from their usage. This separation of concerns makes the code easier to understand, test, and maintain.
2. **Improved Testability**:
   * By injecting dependencies, you can easily replace them with mock objects during unit testing, allowing you to isolate and test individual components without relying on real implementations.
3. **Flexibility and Scalability**:
   * DI makes it easier to swap out implementations of dependencies without modifying the classes that use them. This allows the system to be more flexible and scalable.
4. **Simplified Maintenance**:
   * DI promotes the use of interfaces and abstractions, making it easier to maintain and extend the system. Changes to one part of the system are less likely to affect other parts.
5. **Single Responsibility Principle (SRP)**:
   * DI helps adhere to the Single Responsibility Principle by keeping classes focused on their core functionality, rather than also being responsible for managing dependencies.

**Example in .NET using Dependency Injection**

In .NET, Dependency Injection is widely used and is natively supported by the framework, especially in ASP.NET Core. Here's an example:

csharp

Copy code

// Interface

public interface IEngine

{

void Run();

}

// Implementation

public class DieselEngine : IEngine

{

public void Run()

{

Console.WriteLine("Diesel engine running...");

}

}

// Service

public class Car

{

private readonly IEngine \_engine;

public Car(IEngine engine)

{

\_engine = engine;

}

public void Start()

{

\_engine.Run();

}

}

// Startup Configuration (ASP.NET Core)

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

services.AddScoped<IEngine, DieselEngine>();

services.AddScoped<Car>();

}

public void Configure(IApplicationBuilder app)

{

// Middleware configuration

}

}

// Usage

public class HomeController : Controller

{

private readonly Car \_car;

public HomeController(Car car)

{

\_car = car;

}

public IActionResult Index()

{

\_car.Start();

return View();

}

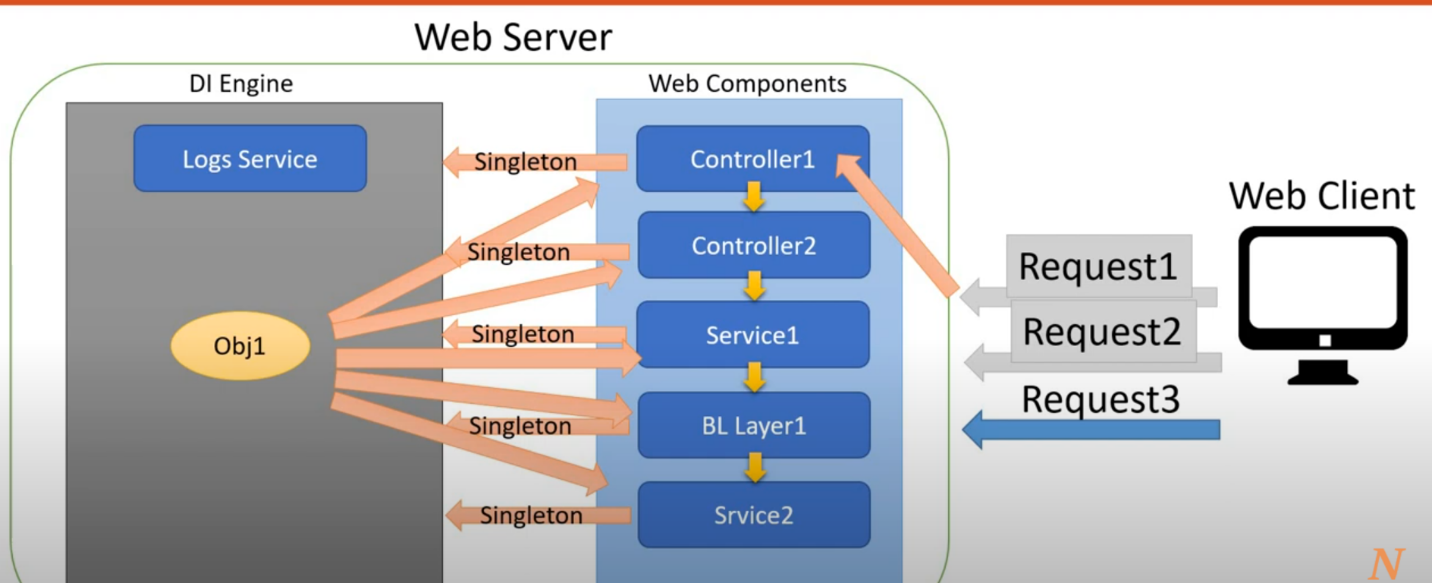
}

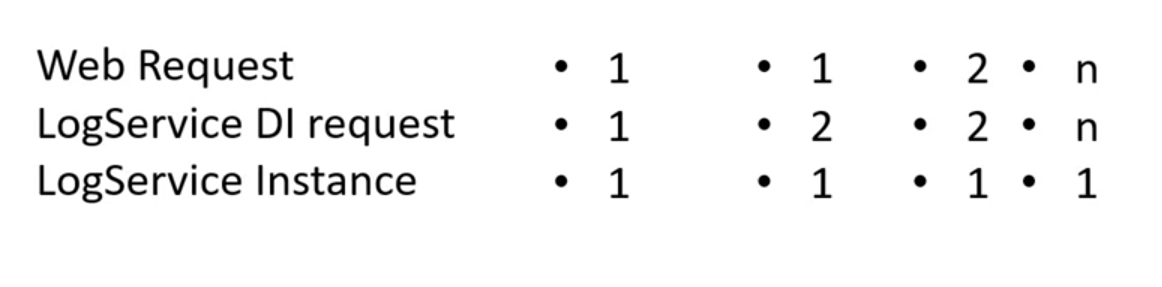
In this example, the DieselEngine is injected into the Car class through its constructor, and the Car class is then injected into the HomeController. The DI container in ASP.NET Core takes care of managing these dependencies.

**Summary**

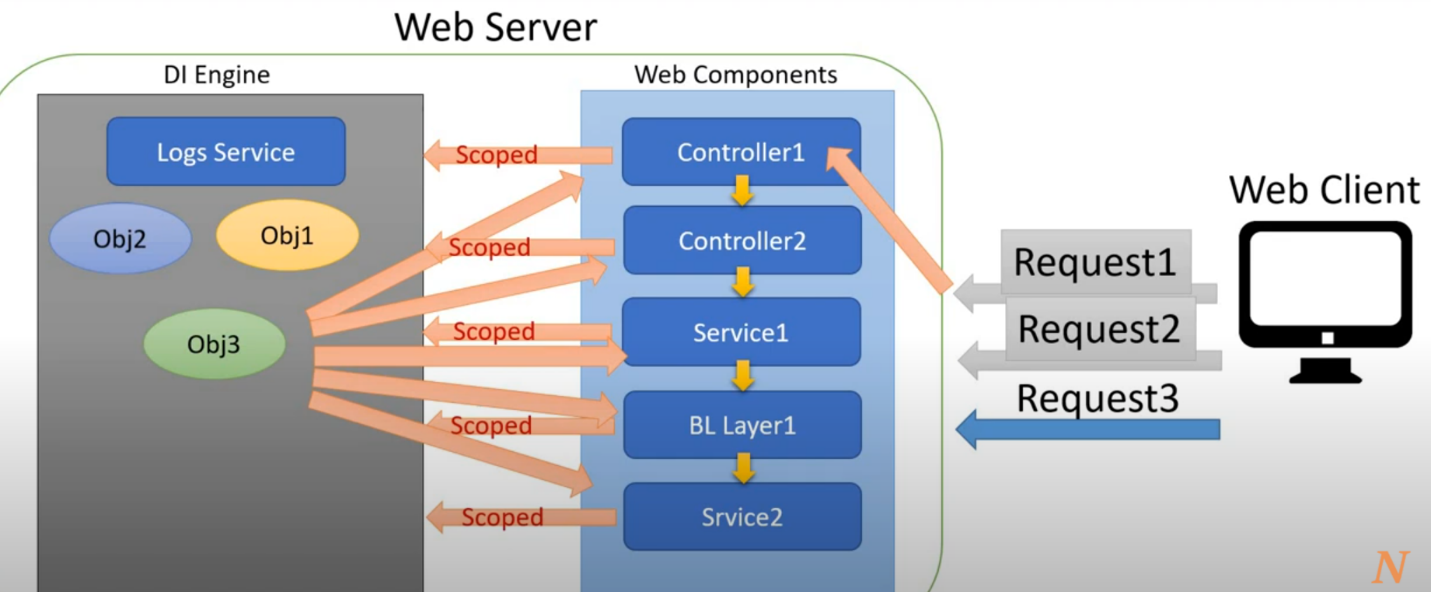
Dependency Injection is a design pattern that helps in decoupling components in a system, making the codebase more modular, testable, and maintainable. It is achieved through various methods like constructor, property, method, and interface injection, and is widely used in modern software development practices.

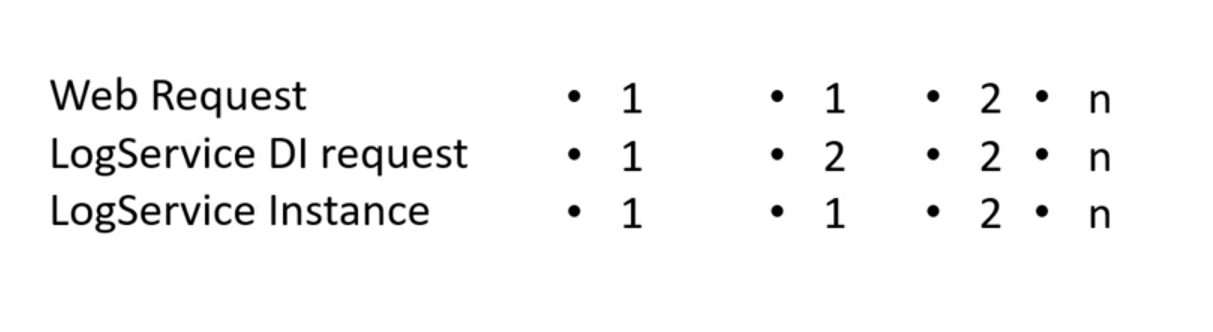
# Add Singleton





# Add Scoped





# Add Transient

