##### **Why do we need the Dependency Injection ?**

The Dependency Injection Design Pattern is one of the most used design Patterns in real-time applications.

 It is important to know why and how to use the Dependency Injection Design Pattern in C#

The Dependency Injection is a design pattern that allows us to develop loosely coupled software components.

In other words, we can say that this design pattern is used to reduce the tight coupling between the software components. As a result, we can easily manage future changes and other complexity in our application.

Before understanding the **Dependency Injection Design Pattern using C#**, first, we need to understand **what is tight coupling** and**what is loose coupling** in software development.

##### **What is Tight Coupling in Software Design?**

Tight coupling means classes and objects are dependent on each other. That means when a class is dependent on another concrete class, then it is said to be a tight coupling between these two classes. In that case, if we change the dependent object, then we also need to change the classes where this dependent object is used. If your application is a small one, then it is not that difficult to handle but if you have a big enterprise-level application, then its really very difficult to handle to make these changes.

##### **What is Loose Coupling in Software Design?**

Loosely coupling means two objects are independent of each other. That means if we change one object then it will not affect another object. The loosely coupled nature of software development allows us to manage future changes easily and also allows us to manage the complexity of the application.

##### **What is Dependency Injection (DI) Design Pattern?**

The Dependency Injection is a process in which we are injecting the object of a class into a class that depends on it.

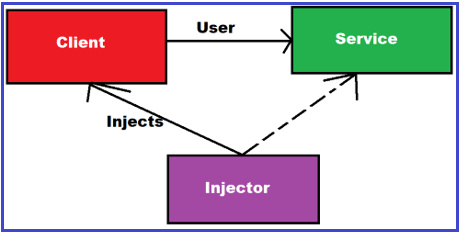
The Dependency Injection is the most commonly used design pattern nowadays to remove the dependencies between the objects that allow us to develop loosely coupled software components.

Dependency Injection (DI) is a design pattern used to implement IoC. It allows the creation of dependency objects outside of a class and provides those objects to a class in different ways. Using DI, we move the creation and binding of the dependent objects outside of the class that depends on them.

Dependency Injection pattern involves 3 types of classes:

1. **Client Class:** The Client class (dependent class) is a class that depends on the service class.
2. **Service Class:** The Service class (dependency) is a class that provides service to the client class.
3. **Injector Class:** The Injector class injects the service class object into the client class.

For better understanding, please have a look at the following diagram.



As you can see above in the above diagram, the injector class creates an object of the service class and injects that object to a client class. In this way, the Dependency Injection pattern separates the responsibility of creating an object of the service class out of the client class.

##### **Different Types of Dependency Injection in C#?**

The injector class injects the dependency object to a class in three different ways. They are as follows.

**Constructor Injection:** When the Injector injects the dependency object (i.e. service) through the client class constructor, then it is called as Constructor Injection.

**Property Injection:** When the Injector injects the dependency object (i.e. service) through the public property of the client class, then it is called as Property Injection. This is also called as the Setter Injection.

**Method Injection:** When the Injector injects the dependency object (i.e. service) through a public method of the client class, then it is called as Method Injection. In this case, the client class implements an interface that declares the method(s) to supply the dependency object and the injector uses this interface to supply the dependency object (i.e. service) to the client class.

##### **Dependency Injection in ASP.NET Core**

ASP.NET Core Provides inbuilt support for Dependency Injection.

The ASP.NET Core Framework is designed from scratch to support inbuilt support for Dependency Injection. The ASP.NET Core Framework injects objects of dependency classes through constructor or method by using a built-in IoC (Inversion of Control) container.

The built-in container is represented by IServiceProvider implementation that supports constructor injection by default. The types (classes) managed by built-in IoC containers are called services.

##### **Types of Services in ASP.NET Core**

There are two types of services in ASP.NET Core. They are as follows:

1. **Framework Services:** Services that are a part of the ASP.NET Core framework such as IApplicationBuilder, IWebHostEnvironment, ILoggerFactory, etc.
2. **Application Services:** The services (custom types or classes) which you as a programmer create for your application.

In order to let the IoC container automatically inject our application services, we first need to register them with the IoC container.

**How to register a Service with ASP.NET Core Dependency Injection Container?**

We need to register a service with ASP.NET Core Dependency Injection Container within the ConfigureServices() method of the Startup class.

**Note*: Before we discuss how to register a service with the Dependency Injection Container, it is important to understand the lifetime of service. When a class receives the dependency object through dependency injection, then whether the instance it receives is unique to that instance of the class or not depends on the lifetime of the service. Setting the lifetime of the dependency object determines how many times the dependency object needs to be created.***

##### **ASP.NET Core Provides  different methods to register a service with Dependency Injection Contains**

The ASP.NET core provides 3 methods to register a service with the ASP.NET Core Dependency Injection container as follows. The method that we use to register a service will determine the lifetime of that service.

1. **Singleton**
2. **Transient**
3. **Scoped**

**Singleton:** In this case, the IoC container will create and share a single instance of a service object throughout the application’s lifetime.

**Transient:** In this case, the IoC container will create a new instance of the specified service type every time you ask for it.

**Scoped:** In this case, the IoC container will create an instance of the specified service type once per request and will be shared in a single request.

**Note:** The Built-in IoC container manages the lifetime of a registered service. It automatically disposes of a service instance based on the specified lifetime.

##### **Extension Methods for Registration**

ASP.NET Core framework includes extension methods for each types of lifetime; AddSingleton(), AddTransient() and AddScoped() methods for singleton, transient and scoped lifetime respectively. The following example shows the ways of registering types (service) using extension methods.

**public** **void** ConfigureServices**(**IServiceCollection services**)**

**{**

//Adding MVC Service. Framework Service

services.AddControllersWithViews**()**;

//Application Service

services.AddSingleton**<**IStudentRepository, TestStudentRepository**>()**;

services.AddSingleton**(**typeof**(**IStudentRepository**)**, typeof**(**TestStudentRepository**))**;

services.AddTransient**<**IStudentRepository, TestStudentRepository**>()**;

services.AddTransient**(**typeof**(**IStudentRepository**)**, typeof**(**TestStudentRepository**))**;

services.AddScoped**<**IStudentRepository, TestStudentRepository**>()**;

services.AddScoped**(**typeof**(**IStudentRepository**)**, typeof**(**TestStudentRepository**))**;

**}**

##### **When to use what?**

In real-time applications, you need to register the components such as application-wide configuration as Singleton. The Database access classes like Entity Framework contexts are recommended to be registered as Scoped so that the connection can be re-used. If you want to run anything in parallel then it is better to register the component as Transient.

###### **So, in short:**

**AddSingleton():**When we use the AddSingleton() method to register a service, then it will create a singleton service. It means a single instance of that service is created and that singleton instance is shared among all the components of the application that require it. That singleton service is created when we requested for the first time.

**AddScoped():**Scoped means instance per request. When we use the AddScoped() method to register a service, then it will create a Scoped service. It means, an instance of the service is created once per each HTTP request and uses that instance in other calls of the same request.

**AddTransient():**When we use the AddTransient() method to register a service, then it will create a Transient service. It means a new instance of the specified service is created each time when it is requested and they are never shared.

##### **What are the advantages of using ASP.NET Core Dependency Injection?**

The ASP.NET Core Dependency Injection allows us to develop loosely coupled software components. Using the ASP.NET Core Dependency Injection, it is very easy to swap with a different implementation of a component.