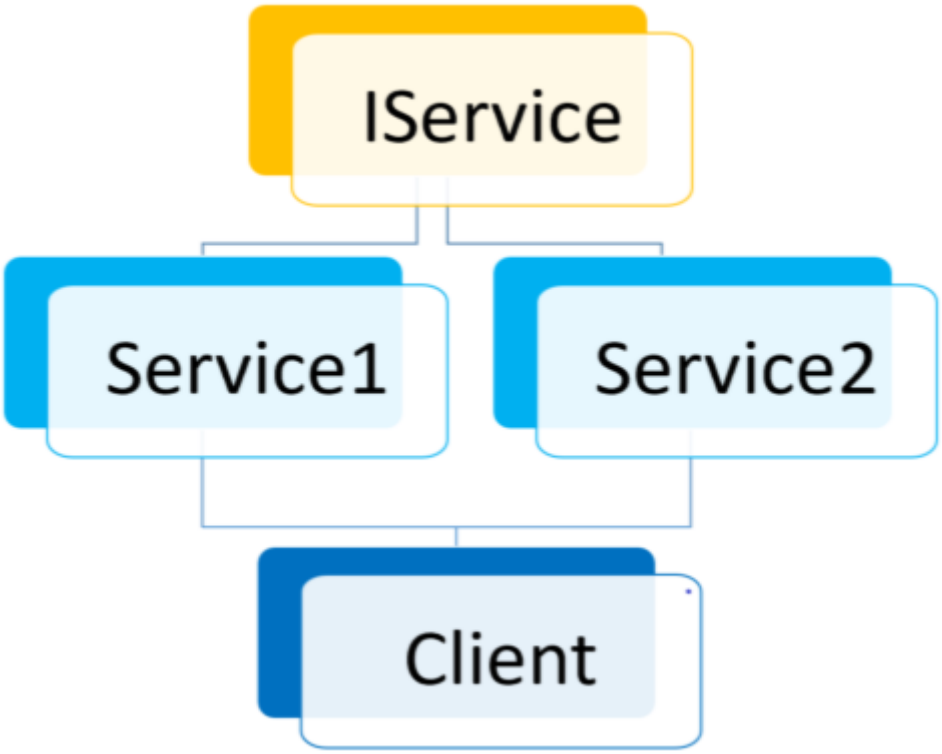
Dependency Injections

Dependency Injection (DI) is a software design pattern that allows us to develop loosely coupled code. DI is a great way to reduce tight coupling between software components. DI also enables us to better manage future changes and other complexity in our software. The purpose of DI is to make code maintainable.

The Dependency Injection pattern uses a builder object to initialize objects and provide the required dependencies to the object means it allows you to "inject" a dependency from outside the class.

For example, Suppose your Client class needs to use two service classes, then the best you can do is to make your Client class aware of abstraction i.e. IService interface rather than implementation i.e. Service1 and Service2 classes. In this way, you can change the implementation of the IService interface at any time (and for how many times you want) without changing the client class code.



We can modify this code by following the Dependency Injection implementation ways. We have the following different ways to implement DI :

Constructor Injection

1. This is a widely used way to implement DI.

2. Dependency Injection is done by supplying the DEPENDENCY through the class's constructor when creating the instance of that class.
3. Injected component can be used anywhere within the class.
4. Recommended to use when the injected dependency, you are using across the class methods.
5. It addresses the most common scenario where a class requires one or more dependencies.

```
public interface IService {

 void Serve();

}

public class Service1 : IService {

 public void Serve() { Console.WriteLine("Service1 Called"); }

}

public class Service2 : IService {

 public void Serve() { Console.WriteLine("Service2 Called"); }

}

public class Client {

 private IService _service;

 public Client(IService service) {

 this._service = service;

 }

 public ServeMethod() { this._service.Serve(); }

}

class Program

{

 static void Main(string[] args)

 {

 //creating object

 Service1 s1 = new Service1();
```

```
 //passing dependency

 Client c1 = new Client(s1);

 //TO DO:



 Service2 s2 = new Service2();

 //passing dependency

 c1 = new Client(s2);

 //TO DO:

 }

}
```

The Injection happens in the constructor, bypassing the Service that implements the IService Interface. The dependencies are assembled by a "Builder" and Builder responsibilities are as follows:

1. Knowing the types of each IService
2. According to the request, feed the abstract IService to the Client

Property/Setter Injection

1. Recommended using when a class has optional dependencies, or where the implementations may need to be swapped.
2. Different logger implementations could be used in this way.
3. Does not require the creation of a new object or modifying the existing one. Without changing the object state, it could work.

```
public interface IService {

 void Serve();

}

public class Service1 : IService {

 public void Serve() { Console.WriteLine("Service1 Called"); }

}

public class Service2 : IService {

 public void Serve() { Console.WriteLine("Service2 Called"); }
```

```
}
public class Client {
 private IService _service;
 public IService Service {
 set { this._service = value; }
 }
 public ServeMethod() { this._service.Serve(); }
}
class Program
{
 static void Main(string[] args)
 {
 //creating object
 Service1 s1 = new Service1();

 Client client = new Client();
 client.Service = s1; //passing dependency
 //TO DO:

 Service2 s2 = new Service2();
 client.Service = s2; //passing dependency
 //TO DO:
 }
}
```

Method Injection

1. Inject the dependency into a single method and generally for the use of that method.

2. It could be useful, where the whole class does not need the dependency, only one method having that dependency.
3. This is the way is rarely used.

```
public interface IService {

 void Serve();

}

public class Service1 : IService {

 public void Serve() { Console.WriteLine("Service1 Called"); }

}

public class Service2 : IService {

 public void Serve() { Console.WriteLine("Service2 Called"); }

}

public class Client {

 private IService _service;

 public void Start(IService service) {

 service.Serve();

 }

}

class Program

{

 static void Main(string[] args)

 {

 //creating object

 Service1 s1 = new Service1();



 Client client = new Client();

 client.Start(s1); //passing dependency
```

```
 //TO DO:



 Service2 s2 = new Service2();

 client.Start(s2); //passing dependency

 }

}
```

Advantages of Dependency Injection

1. Reduces class coupling
2. Increases code reusability
3. Improves code maintainability
4. Make unit testing possible

DI Container

The recommended way to implement DI is, you should use DI containers. If you compose an application without a DI CONTAINER, it is like a `POOR MAN'S DI`. If you want to implement DI within your ASP.NET MVC application using DI container, please do refer Dependency Injection in ASP.NET MVC using Unity IoC Container.

Read More Articles Related to Dependency Injection