Name :- Bhakti Chevli

Topic:Dependency Injection

Date :- 14 Dec , 2020

Rno: 14

1:- What is Dependency Injection?

Ans:-

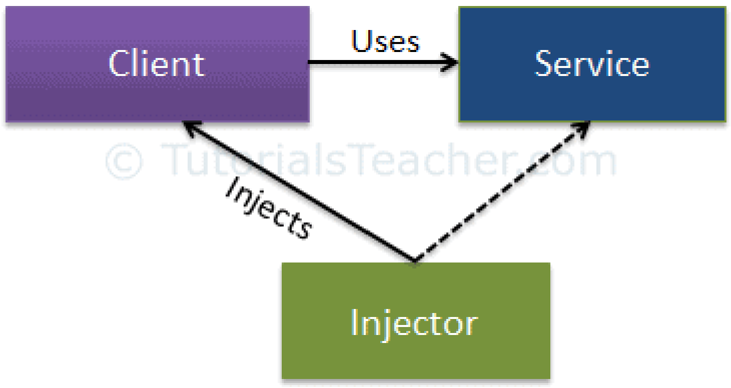
ASP.NET Core supports the dependency injection (DI) software design pattern, which is a technique for achieving Inversion of Control (IoC) between classes and their dependencies.

The Dependency Injection pattern involves 3 types of classes.

1. **Client Class:** The client class (dependent class) is a class which 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.

The following figure illustrates the relationship between these classes:



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

❖ Types of Dependency Injection:-

ConstructorInjection: In the constructor injection, the injector supplies the service (dependency) through the client class constructor.

Property Injection: In the property injection (aka the Setter Injection), the injector supplies the dependency through a public property of the client class.

Method Injection: In this type of injection, the client class implements an interface which declares the method(s) to supply the dependency and the injector uses this interface to supply the dependency to the client class.

2:- How to use Dependency Injection in c# application?

Ans :-

```
public class ClientBusinessLogic
{
IClientDataAccess _dataAccess;

   public
ClientBusinessLogic(IClientDataAccessclientDataAccess)
   {
      _dataAccess = clientDataAccess;
   }

   public ClientBusinessLogic()
   {
      _dataAccess = new ClientDataAccess();
   }

   public string ProcessClientData(int id)
   {
      return _dataAccess.GetClientName(id);
   }
}

public interface IClientDataAccess
{
   string GetClientName(int id);
}

public class ClientDataAccess: IClientDataAccess
{
   public ClientDataAccess()
   {
   }
```

```
    public string GetClientName(int id)
    {
        return "Client Name";
    }
}


public class ClientService
{
ClientBusinessLogic _clientBL;

    public ClientService()
    {
        _clientBL = new ClientBusinessLogic(new
ClientDataAccess());
    }

    public string GetClientName(int id) {
        return _clientBL.ProcessClientData(id);
    }
}
```

Ques 3:-Explain usage of Dependency Injection importance with example.

Ans:-

- 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.

- The problem with the example is that we used `DataAccessFactory` inside the `ClientBusinessLogic` class. So, suppose there is another implementation of `IClientDataAccess` and we want to use that new class inside `ClientBusinessLogic`. Then, we need to change the source code of the `ClientBusinessLogic` class as well. The Dependency injection pattern solves this problem by injecting dependent objects via a constructor, a property, or an interface.

❖ Constructor Injection :-

As mentioned before, when we provide the dependency through the constructor, this is called a constructor injection.

```
public class CustomerBusinessLogic
{
    ICustomerDataAccess _dataAccess;
    public    CustomerBusinessLogic(ICustomerDataAccess
custDataAccess)
    {
        _dataAccess = custDataAccess;
    }

    public CustomerBusinessLogic()
    {
        _dataAccess = new CustomerDataAccess();
    }

    public string ProcessCustomerData(int id)
    {
        return _dataAccess.GetCustomerName(id);
    }
}

public interface ICustomerDataAccess
{
    string GetCustomerName(int id);
}

public class CustomerDataAccess: ICustomerDataAccess
{
    public CustomerDataAccess()
    {
    }

    public string GetCustomerName(int id)
    {
        //get  the  customer  name  from  the  db  in  real
application
        return "Dummy Customer Name";
    }
}
```

In the above example, `ClientBusinessLogic` includes the constructor with one parameter of type `IClientDataAccess`. Now, the calling class must inject an object of IClientDataAccess.

```
public class CustomerService
{
    CustomerBusinessLogic _customerBL;

    public CustomerService()
    {
        _customerBL = new CustomerBusinessLogic(new
CustomerDataAccess());
    }

    public string GetCustomerName(int id) {
        return _customerBL.ProcessCustomerData(id);
    }
}
```

As you can see in the above example, the `ClientomerService` class creates and injects the `ClientomerDataAccess` object into the `ClientBusinessLogic` class. Thus, the `ClientBusinessLogic` class doesn't need to create an object of `ClientDataAccess` using the `new` keyword or using factory class. The calling class (ClientService) creates and sets the appropriate DataAccess class to the `ClientBusinessLogic` class. In this way, the `ClientBusinessLogic` and `ClientDataAccess` classes become "more" loosely coupled classes.

❖ Property Injection :-

In the property injection, the dependency is provided through a public property.

```
public class ClientBusiLogic

{

    public ClientBusiLogic()

    {

    }


    public string GetClientName(int id)

    {

        return DataAccess.GetClientName(id);

    }


    public IClientDataAccessDataAccess { get; set; }

}


public class ClientService

{

ClientBusiLogic _clientBL;


    public ClientService()

    {

        _clientBL = new ClientBusiLogic();
        _clientBL.DataAccess = new ClientDataAccess();

    }


    public string GetClientName(int id) {

        return _clientBL.GetClientName(id);

    }

}
```

As you can see above, the `ClientomerBusinessLogic` class includes the public property named `DataAccess`, where you can set an instance of a class that implements `IClientomerDataAccess`.
So, `ClientomerService` class creates and sets `ClientDataAccess` class using this public property.

❖ Method Injection:-

In the method injection, dependencies are provided through methods. This method can be a class method or an interface method.

```
interface IDataAccessDependency

{

    void SetDependency(IClientDataAccessclientDataAccess);

}
```

```
public class ClientBusiLogic : IDataAccessDependency
{
IClientDataAccess _dataAccess;

    public ClientBusiLogic()
    {
    }

    public string GetClientName(int id)
    {
        return _dataAccess.GetClientName(id);
    }

    public void SetDependency(IClientDataAccessclientDataAccess)
    {
        _dataAccess = clientDataAccess;
    }
}

public class ClientService
{
ClientBusiLogic _clientBL;
```

```
    public ClientService()

    {

        _clientBL = new ClientBusiLogic();

        ((IDataAccessDependency)_clientBL).SetDependency(new
ClientDataAccess());

    }



    public string GetClientName(int id) {

        return _clientBL.GetClientName(id);

    }

}
```

In the above example, the `ClientBusinessLogic` class implements the `IDataAccessDependency` interface, which includes the `SetDependency()` mehtod. So, the injector class `ClientService` will now use this method to inject the dependent class (ClientDataAccess) to the client class.