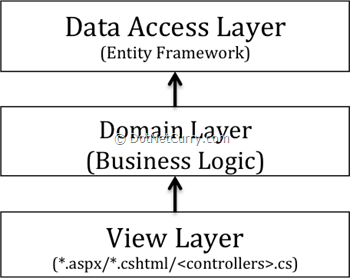
As Developers we may have come across the term ‘Dependency Injection’ as a design pattern that helps develop maintainable and de-coupled code. DI is also a way to implement the ‘D’ in [SOLID principles](http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod). (SOLID as stated by Uncle Bob stands for: Single responsibility, Open closed principle, Liskov substitution principle, Interface segregation principle and Dependency Inversion). Often the association of 'Design Pattern' to anything that we have not dealt with before, typically means it goes into our 'To Do' bucket which becomes very dusty over time. Time to blow the dust away!

Dependency Injection in ASP.NET MVC (C#)

In this article, we are going to look at a little bit of the theory and then do a code walkthrough to understand basics of DI. By the end of the article, we should be clear about the pattern and **how to apply Dependency Injection in an ASP.NET MVC applicatio**n.

**Dependency Injection** is often associated with IoC (Inversion of Control) containers. We will not look at any IoC containers. We leave it for the next article in the series.

To get an idea of why we should adopt Dependency Injection (DI), we will first look at an abstract example without DI. Then we'll convert it into one with DI. Let us start off with a standard three-layer web application design. Figure 1 depicts a standard three-layer application.

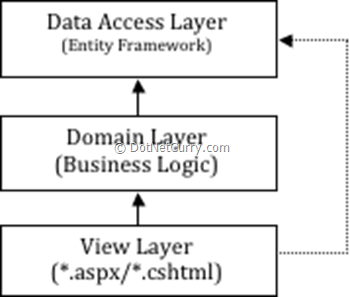


The arrows above depict direction of coupling, as in the View layer is 'dependent' or 'has reference of' the Domain Layer and the Domain Layer is 'dependent' on the Data Access Layer. This is a common design where each layer represents a project in Visual studio.

On the outset, it looks a perfectly okay design approach. The way to use this would be the View layer code behind (or Controller classes in MVC) would instantiate a Domain Layer class and the domain layer would instantiate a Data access layer class and the calls from the View layer would get chained like this. Initially this seems harmless but programming against concrete instances actually makes our code very tightly coupled and highly 'dependent' on each other. Even though putting data access code separate from business logic code offers some immunity, it still does not give you any design benefit if in the future you had to change data sources say from a local source to a cloud source.

You will be forced to go and change declaration of every instance of the data access layer. In other words it is not a plug-in architecture.

Things get a little more complicated if you are using Data Access Entities (like the Entity Framework) and performing business manipulations on them. That would require the View layer to be 'aware' of these entities and hence refer to the Data Access Layer directly. The above diagram now becomes as follows:

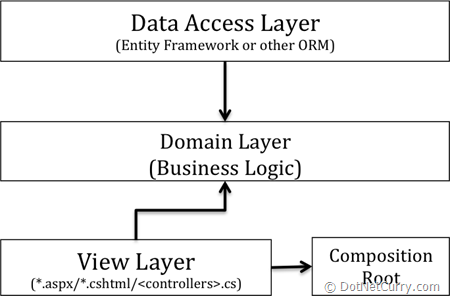


Now this becomes a recipe for disaster because now you have lost control on Data Access and there is nothing stopping from the team newbie calling the data access layer directly and writing a loop in the code behind to manipulate data. Very soon you have your business logic all over the place and your seemingly well thought out layering has gone completely haywire.

**Dependency Injection to the rescue**

The term Dependency Injection, implies sending a dependency (DAL) into a dependent object (Domain Layer) as opposed to the dependent object controlling life cycle of its dependencies.

That would mean, that the domain layer should not instantiate the Data Access layer, the correct data access instance should be passed to it. Who will pass the instance? Is it the View layer? But does that mean View layer will instantiate the Data Access Layer? No. The View Layer will delegate the object instantiation to a special class called 'COMPOSITION ROOT'. The Composition Root will use configuration and a Factory to determine the correct instance. It will create an instance of the dependency (using reflection) and pass it into the View Layer. Since business layer is our glue between UI and persistence, it is okay to instantiate instance of Business layer directly in the view layer. To make the abstraction complete, the business logic layer should define an abstract class or interface for the Data Access layer to implement. This prevents code instantiation of data access layer anywhere in the Business or View layer. Our design diagram now changes to the following



Notice how the Domain Layer is no longer coupled to Data Access Layer in fact, the DAL now refers to the Domain Layer. This is because the Domain Layer defines the Interfaces that are implemented by the DAL Layer.

Okay, this resolves the dependency creation. But now how will the UI layer deal with data? Answer is 'Domain Objects'. The Domain layer will have to define Domain objects that are POCOs (Plain Old CLR Objects). The Repository implementation will translate the persistent entity Objects in DAL layer to Domain POCOs.

If you are ready to give up with the theory, hang in there for one more minute and we'll get to code.

Before we jump into code we should clarify one more term in DI, *Constructor Injection*. For DI to work, the dependencies need to be passed in to the dependent objects. The most common way to pass Dependencies is through constructor injection where in, all dependencies are pass as constructor parameters of the dependent objects. For example, in the above diagram, the composition root can pass instances of the Data Access layer only through the constructor of the Controllers (in View layer) and the View layer can pass that instance to the Domain layer only through the constructor of the Business Logic layer. This has an additional side effect i.e. we cannot use empty constructors for DI layers.