Introduction

The scope of this tutorial is to unravel (hopefully) some of the mistery behind **Dependency Injection**, a widely used pattern, that although its implementation may seem complicated (and it may very well be), the concept on the other hand is relatively simple and quite useful.

Terminology

In the same context you may encounter three different terms that are related but not quite synonyms, and these are:

* Dependency Injection (DI)
* Inversion of Control (IoC)
* Dependency Inversion Principle (DIP)

**Dependency Injection (DI)**

In software engineering, dependency injection is a design pattern whereby one object supplies the dependencies of another object. A dependency is an object that can be used (a service). An injection is the passing of a dependency to a dependent object (a client) that would use it. The service is made part of the client's state. Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

**Dependency injection** is one form of the broader (more general) technique of inversion of control.

The intent behinddependency injection is to decouple objects to the extent that no client code has to be changed simply because an object it depends on needs to be changed to a different one.

**Inversion of Control (IoC)**

In software engineering, inversion of control is a design principle used to increase modularity of the program and make it extensible.  
More specifically this means that objects do not create other objects on which they rely to do their work. Instead, they get the objects that they need from an outside source. So we invert/revert the control (hence the name IoC), by delegating the responsibility of creating objects/instances to code outside the class where we need these dependencies. Fortunately there are dependency injection frameworks (commonly know as IoC containers) that make our life easier by taking care of creating these objects (the term usually used is getting or  resolving instances).   
There are several basic techniques to implement inversion of control (i.e. factory pattern, service locator pattern, dependency injection and so on).

Dependency Inversion Principle Tutorial (DIP)

In object-oriented design, the dependency inversion principle refers to a specific form of decoupling software modules. When following this principle, the conventional dependency relationships established from high-level, policy-setting modules to low-level, dependency modules are reversed, thus rendering high-level modules independent of the low-level module implementation details. The principle states:   
**A.** High-level modules should not depend on low-level modules. Both should depend on abstractions.  
**B.** Abstractions should not depend on details. Details should depend on abstractions.  
In the practical sense this means that you should make use of interfaces when applying this principle as illustrated in the example.

Example of Dependency Injection in ASP.NET MVC

For this tutorial, let`s consider a simple example where we want to save something to the database and take for instance the user creation in an [**ASP.NET**](https://assist-software.net/portfolio/technologies-skills/asp.net)**MVC**application. So first we need a UserController class and a Create action method (overlooked return statement for practical reasons).

[?](https://assist-software.net/blog/dependency-injection-aspnet-mvc-tutorial)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class UserController : Controller      {          public ActionResult Create(User user)          {              // save user to database          }      } |

From here, as usual, we have several options. The worst would be to write here the actual code that saves the user to the database. Adding a separate method in UserController that does the saving would also be bad because:

* Controllers should have mostly action methods and no business logic
* It violates the Single Responsibility Principle (aside from its intended responsibility we would charge the controller with the responsibility of saving to the database)

The solution would be to create a separate class that has the responsibility of working with the database.

[?](https://assist-software.net/blog/dependency-injection-aspnet-mvc-tutorial)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class UserController : Controller      {          public ActionResult Create(User user)          {              var repository = new Repository();              repository.Save(user);          }      } |

This solves the SRP issue described above but by creating a Repository object in UserController we get a dependency that is hard to change (tight coupling). Suppose we would also want to update existing user information.

[?](https://assist-software.net/blog/dependency-injection-aspnet-mvc-tutorial)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class UserController : Controller      {          public ActionResult Create(User user)          {              var repository = new Repository();              repository.Save(user);          }            public ActionResult Edit(User user)          {              var repository = new Repository();              repository.Update(user);          }      } |

So now we have two actions with a dependency on Repository but there could also be different entities (i.e. Product, ShoppingCart, Order, etc.) in different controllers, all depending on Repository. Imagine a scenario where we would like to use a different Repository (i.e. MySqlRepository, MongoDbRepository, etc.) or even just provide mock objects for unit testing. We would then have to change every line and replace new Repository() with the one in question. Of course we do not want to do this. First, this is no fun. Second, this is error-prone. Third, this is stupid, repetitive work for a trained monkey. So what do we do?   
This is where Dependency Injection comes in. The solution would be to inject a Repository dependency in the UserController constructor (there is also setter injection and property injection but constructor injection is the most common). By injecting the dependencies we achieve great flexibility in our application since we can now swap, decorate and intercept dependencies without the consuming class knowing. In other words, changing the Repository does not imply any change in UserController (or any other for that matter).

[?](https://assist-software.net/blog/dependency-injection-aspnet-mvc-tutorial)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class UserController : Controller      {          private readonly Repository \_repository;            public UserController(Repository repository)          {              \_repository = repository;          }            public ActionResult Create(User user)          {              \_repository.Save(user);          }            public ActionResult Edit(User user)          {              \_repository.Update(user);          }      } |

Now, when we change the Repository, we need only change the constructor parameter and the field. But we can take this even further by following the **Dependency Inversion Principle** through shared abstractions which will truly imply no change at all in the controllers. Practically speaking we could use an interface that every repository will need to implement. So instead of Repository we use IRepository.

[?](https://assist-software.net/blog/dependency-injection-aspnet-mvc-tutorial)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class UserController : Controller      {          private readonly IRepository \_repository;            public UserController(IRepository repository)          {              \_repository = repository;          }            public ActionResult Create(User user)          {              \_repository.Save(user);          }            public ActionResult Edit(User user)          {              \_repository.Update(user);          }      } |

As mentioned earlier there are DI frameworks, also know as IoC containers that take care of resolving the instances. For example, using Castle Windsor IoC container we need only register / install the services (in our case the IRepository service). This would look something like this:

[?](https://assist-software.net/blog/dependency-injection-aspnet-mvc-tutorial)

|  |  |
| --- | --- |
| 1  2  3  4 | public void Install(IWindsorContainer container, IConfigurationStore store)          {  container.Register(Component.For<IRepository>().ImplementedBy<Repository>().LifeStyle.Transient);          } |

Now if we want to change the Repository we change it only here, in one place. Of course that in order for this to work you also need to add a custom ControllerFactory and install your IoC container registrations but this is beyond the scope of this tutorial which is meant to focus more on the concept and the advantages you get by using Dependency Injection.

Pros

* loosely coupled code
* reusable code
* testable code
* readable code
* increases modularity and adds extensibility

Cons

* adds a degree of complexity
* the learning curve may be an issue

Conclusion

Even if you never plan to change the injected service (in this case the Repository) and you never plan to do Unit Testing **I still highly recommend you to use Dependency Injection** whenever is suited and this**tutorial**is meant to help you succeed easier with your plans (i.e. in controllers as illustrated in the example).

You can read more about Dependency Injection at <http://stackoverflow.com/questions/14301389/why-does-one-use-dependency-injection>

## Dependency injection in MVC using Unity IoC container

If you have experience in software design and follow (or at least try to follow) perfect design pattern and principals, then dependency injection, de-couple architecture, and Inversion of Control (IoC) are very common terms for you. You might have hands on experience with the implementation of dependency injection in various programming languages (with or without any IoC container). In this article, we will discuss the relationship between dependency injection and IoC containers and then we will see how IoC containers and dependency injection fit side by side.

## What is IoC container ?

The DI container or IoC container is a software framework used to create dependencies and inject them automatically when required. Truth to be told, some dependency containers work like magic when it needs to resolve dependency. There are many such frameworks available in the market to simplify our work and here are a few which work in the .NET environment. So the choice is completely yours, they each have their own benchmarks and performance so choose wisely depending on your project and requirements. As this is article is not dedicated to the IoC container, I will not stretch the discussion on it too far. In this article we will use Unity as an IoC container to resolve dependency.

## Let’s Implement Controllers with Dependency

As the heading indicates, we will implement MVC architecture to help understand the concept and then we will create two controllers with some of the dependency in each of them and see how we can solve the dependency using an IoC container. Here is the code of the first controller. The implementation is very simple, we have created an Ilogger interface and there are two implementations of ILogger: one is ActualLogger which is not yet completed, and as substitute we have implemented another class, FakeLoggerm which we will use in code development.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace MVCIoC.Controllers

{

public interface ILogger

{

Boolean Log();

}

public class ActualLogger : ILogger

{

public Boolean Log()

{

throw new NotImplementedException();

}

}

public class FakeLogger : ILogger

{

public Boolean Log()

{

return true;

}

}

public class HomeController : Controller

{

public readonly ILogger logger = null;

public HomeController(ILogger tmpLogger)

{

logger = tmpLogger;

}

public ActionResult Index()

{

*//perform logging information*

logger.Log();

return View();

}

}

}

So the idea is that, if our actual implementation is in the development phase, we can still continue our controller development using some kind of substitution and in time, we can change it without affecting the existing code. So we are seeing that the Home controller has one parameterized constructor. As a parameter it will take the actual implementation of a dependent object. Let’s create another controller; the implementation will be exactly the same as the above one. I just wanted to show how to solve dependency in multiple controllers.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace MVCIoC.Controllers

{

public interface IService

{

Boolean Call();

}

public class RealService : IService

{

*//Real service is not developed yet.*

public bool Call()

{

throw new NotImplementedException();

}

}

public class FakeService : IService

{

public bool Call()

{

return true;

}

}

public class ValueController : Controller

{

public readonly IService service = null;

public ValueController(IService tmpService)

{

service = tmpService;

}

public ActionResult Index()

{

*//User service to implement business logic*

service.Call();

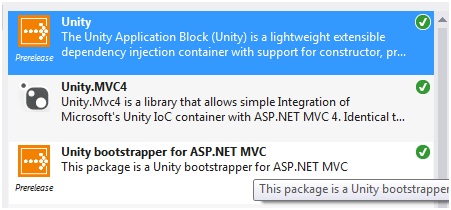
return View();

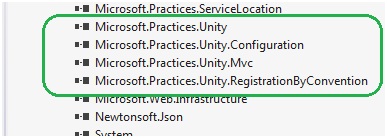
}

}

}

OK, we have setup everything. Now we will give a reference of Unity Framework. If you are using Visual Studio then you can use NuGet Package Manager to give reference. I have given the following references as an example:





Now we will implement our own controller factory class from the DefaultControllerFactory class. There are other options you can implement as well. For example, IController performs the same operation. Here we have defined ControllerFactory and derived from the DefaultControllerFactory class.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

using System;

using System.Collections.Generic;

using System.Linq;

using System.Reflection;

using System.Web;

using System.Web.Mvc;

using System.Web.Routing;

using Microsoft.Practices.Unity;

using MVCIoC.Controllers;

namespace MVCIoC.Models

{

public class ControllerFactory : DefaultControllerFactory

{

protected override IController GetControllerInstance(RequestContext requestContext, Type controllerType)

{

try

{

if (controllerType == null)

throw new ArgumentNullException("controllerType");

if (!typeof(IController).IsAssignableFrom(controllerType))

throw new ArgumentException(string.Format(

"Type requested is not a controller: {0}",

controllerType.Name),

"controllerType");

return MvcUnityContainer.Container.Resolve(controllerType) as IController;

}

catch

{

return null;

}

}

}

public static class MvcUnityContainer

{

public static UnityContainer Container { get; set; }

}

}

You can see that we are resolving the dependency within the try block and it will create a controller object by resolving dependency. In the case of the Unity container, we will create a Bootstrapper class to set all the dependencies in a single file. Have a look at the code below. Though it’s not mandatory to implement Bootstrapper it’s always a good practice to implement it.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using Microsoft.Practices.Unity;

using Microsoft.Practices.Unity.Mvc;

using MVCIoC.Controllers;

namespace MVCIoC.Models

{

public class Bootstrapper

{

public static IUnityContainer Initialise()

{

var container = BuildUnityContainer();

DependencyResolver.SetResolver(new UnityDependencyResolver(container));

return container;

}

private static IUnityContainer BuildUnityContainer()

{

var container = new UnityContainer();

container.RegisterType<IService, FakeService>();

container.RegisterType<ILogger, FakeLogger>();

MvcUnityContainer.Container = container;

return container;

}

}

}

Please notice that we have mapped FakeService and FakeLogger with its interface because our aim is to use the implementation of FakeService and FakeLogger. Once the development of the original implementation is done you can just change in this place, there is no need to touch your controller code. Now the final implementation is registration. We have to register our IoC container in the MVC channel. Just modify your Application\_Start() code as shown below.

Hide   Copy Code

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

RouteConfig.RegisterRoutes(RouteTable.Routes);

*//Initialize IoC container/Unity*

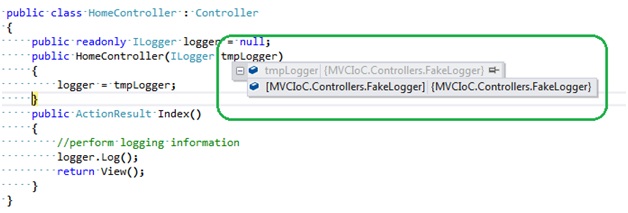
Bootstrapper.Initialise();

*//Register our custom controller factory*

ControllerBuilder.Current.SetControllerFactory(typeof(ControllerFactory));

}

Now, it’s time to run the application. Once we call the home controller you will see that the object of FakeLogger has injected to the constructor of the Home controller.



## Border Line

There are many IoC containers available on the market. You can choose any one of them, and all IoC containers work more or less in the same way.