# [.NET Core] Dependency Injection in ASP .NET Core – “Old but gold”

Inthis article, we will talk about the Dependency Inversion Principle (**DIP**), Inversion of Control (**IoC**) and Dependency Injection (**DI**). And then try to create a simple **DI** implementation with the built-in feature of .NET Core with different ways (Controller Constructor Injection, Controller Method Injection and View Injection).

***Note****: If you already understand what are Dependency Inversion Principle (DIP) , Inversion of Control (Ioc) and Dependency Injection (DI) then feel free to bypass these sections.*

# Dependency Inversion Principle (DIP)

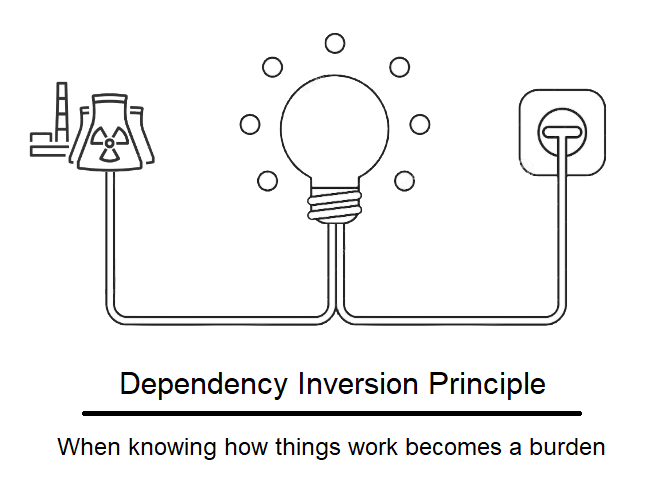


image source: internet

Dependency Inversion Principle is “**D**” in the SOLI**D**acronym. The DIP has been introduced in the 90s by [Robert C Martin](https://en.wikipedia.org/wiki/Robert_C._Martin). [Here](https://drive.google.com/file/d/0BwhCYaYDn8EgMjdlMWIzNGUtZTQ0NC00ZjQ5LTkwYzQtZjRhMDRlNTQ3ZGMz/view) is the original article.

**DIP** is a software design that guides us to write loosely classes. According to the definition of **DIP** on the wiki:

* High-level modules should not depend on low-level modules. Both should depend on abstractions.
* Abstractions should not depend upon details. Details should depend upon abstractions.

Let me give you an example to describe more about the above definitions. Let’s imagine, you need to write a function that allows the system to send an email to the end-user when they finish an order. There are 2 classes should be created, one for ordering and one for sending the email.

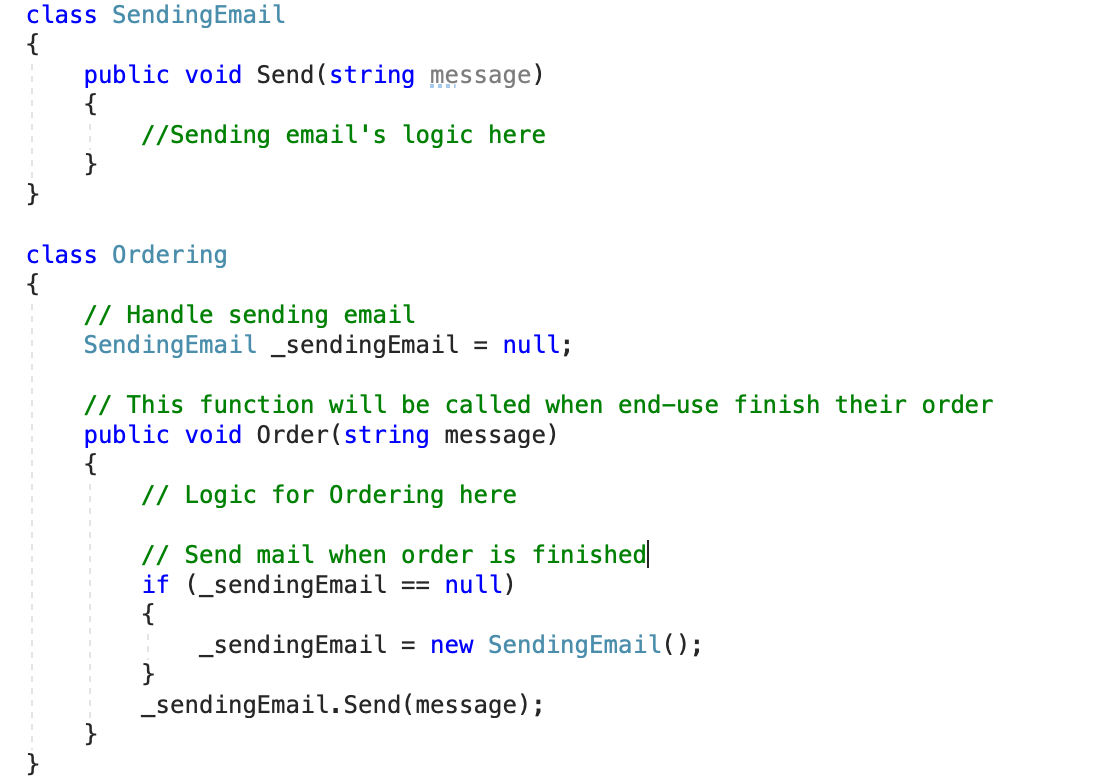


figure 1: SendingEmail and Ordering class

For the first look, there is no problem with the code logic, as “Send” function will be triggered once end-user finish their order. However, it violates **DIP** because the **Ordering** class depended upon the SendingEmail class and SendingEmail class is not an abstraction — it is a concrete class. But what is the problem here? What happens if there is a new requirement from the business team that requires you have to change the communication type to use SMS instead of Email?

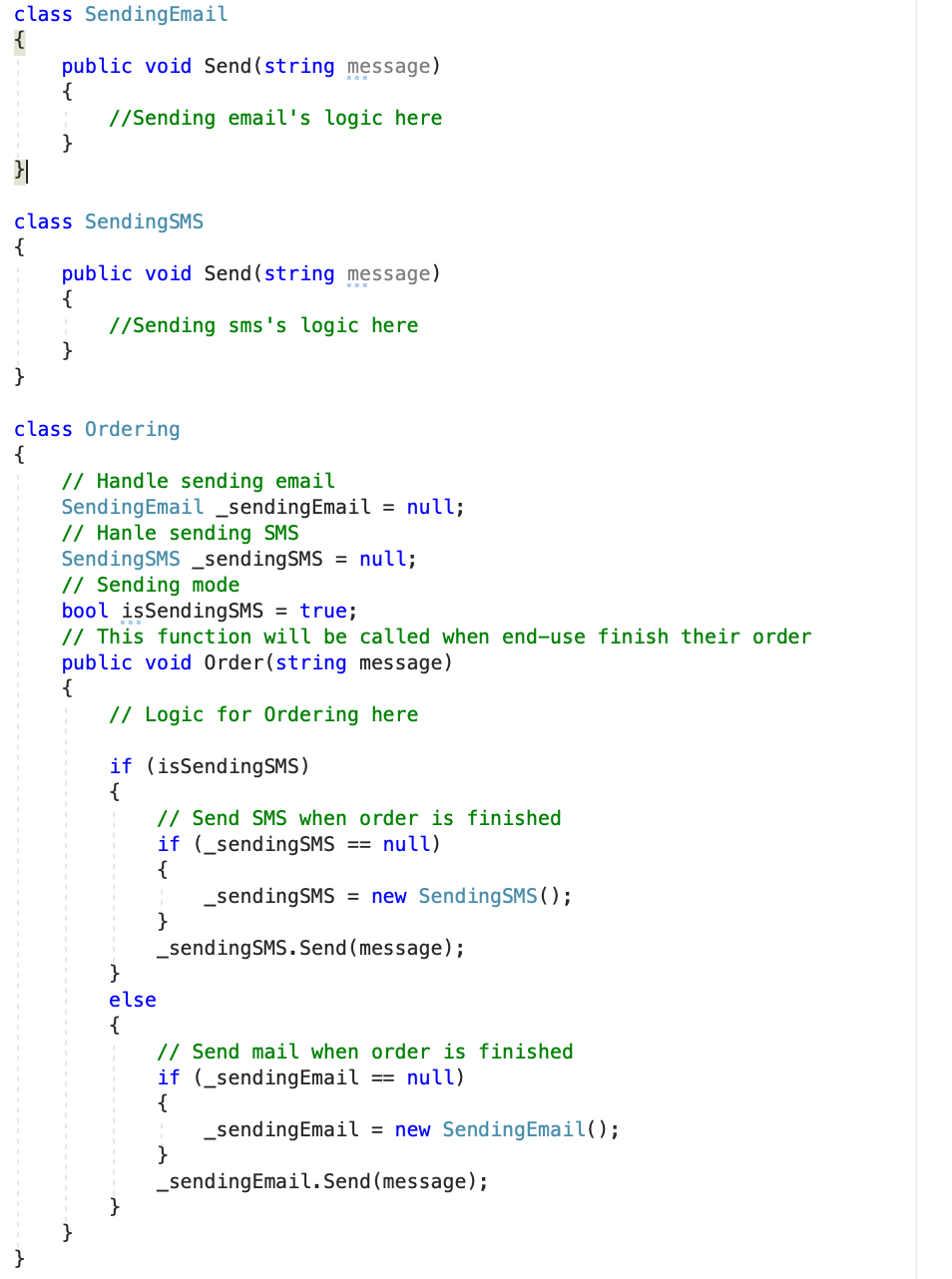


figure 2: Adding more SendingSMS class

As a result, you will have to create more **SendingSMS** class and declare an instance of it within the **Ordering** class. Finally, use “IF ELSE” statement to decide either using SMS or using Email. But it even gets worse, when you have more options to select beside Email and SMS. It means you have to declare more instances of new concrete classes within Ordering class.

The **DIP** says that we need to decouple the system in such a way that the higher-level modules (in this case is **Ordering**) will depend on abstractions and use it instead of concrete classes. The abstraction will be mapped to the actual concrete class that will perform code logic. (You can see it in next sections)

# Inversion of Control (IoC)

Inversion of Control (**IoC**) is the technique that helps us to make higher-level modules to depend on abstractions rather than the concrete implementation of lower-level modules. In other words, it helps to implement the (Dependency Inversion Principle — **DIP**). Let’s come back with the sample that I showed you above and implement **IoC**.

Firstly, we need to create an abstraction that the higher-level — **Ordering** class will depend on.

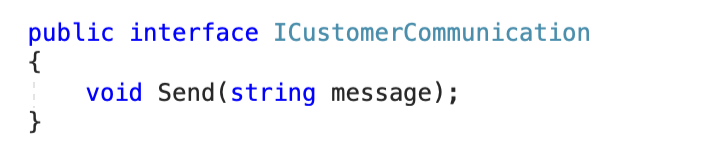


figure 3: ICustomerCommunication interface — an abstraction

And then update “**SendingEmail**” and “**SendingSMS**” class to inherit from **ICustomerCommunication** interface.

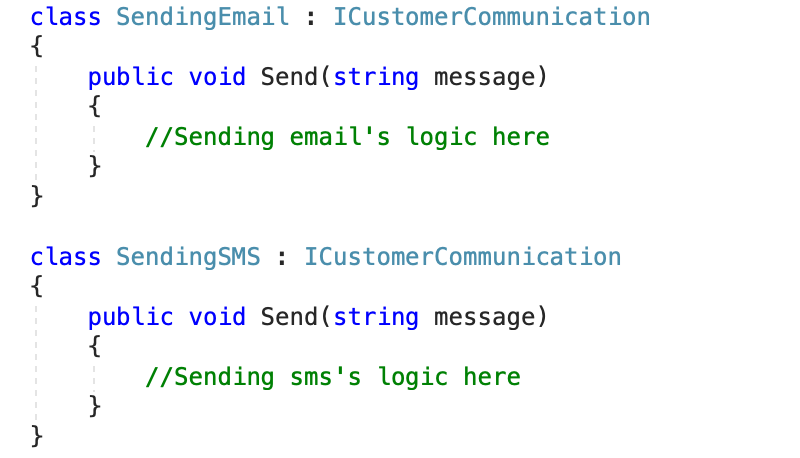


figure 4: Update SendingEmail and SendingSMS class

Now let’s change the higher-level module — **Ordering** class to use this abstraction rather than the lower level concrete class.

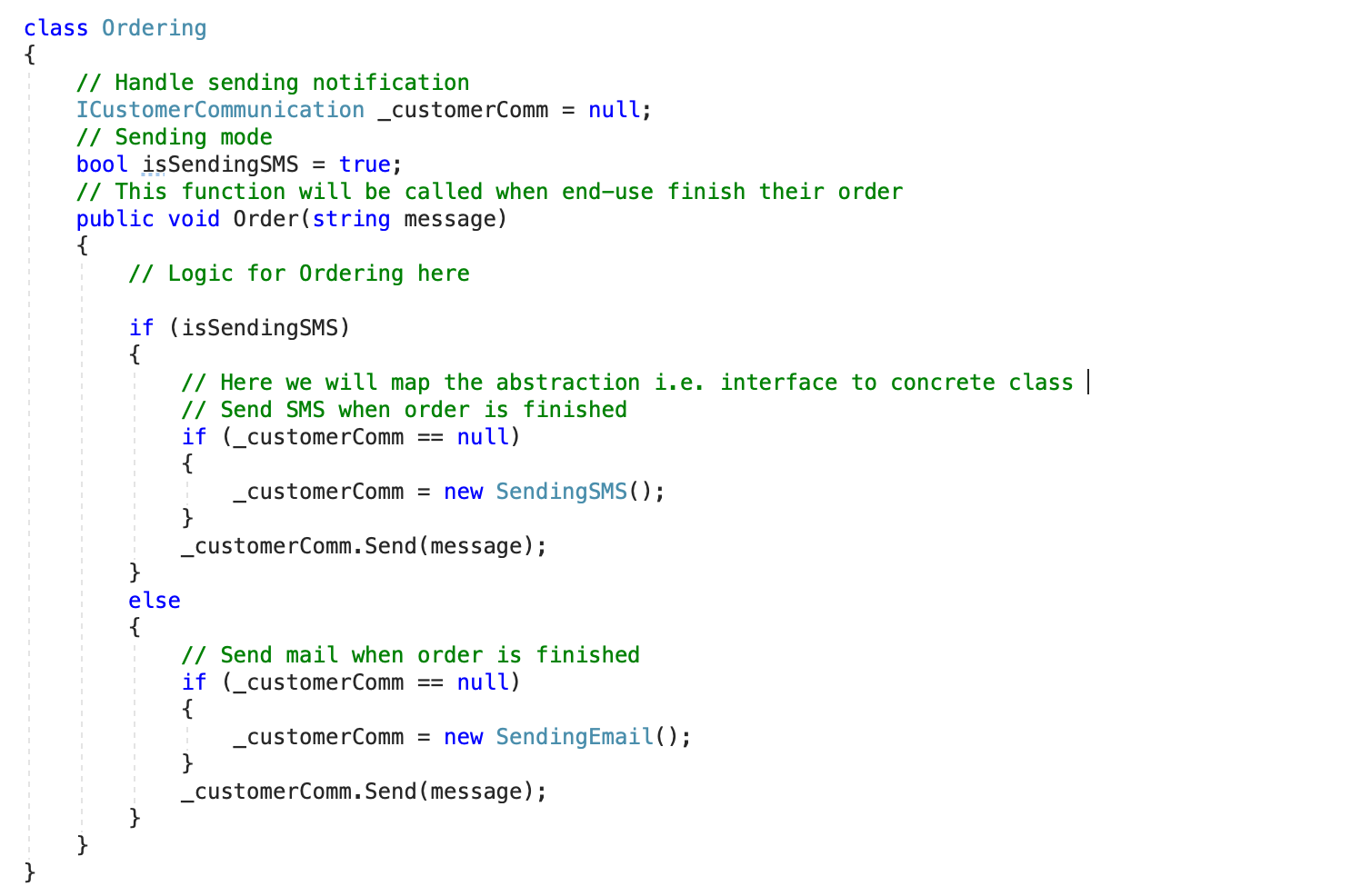


figure 5: Update higher-level class

Finally, the design will look like this:

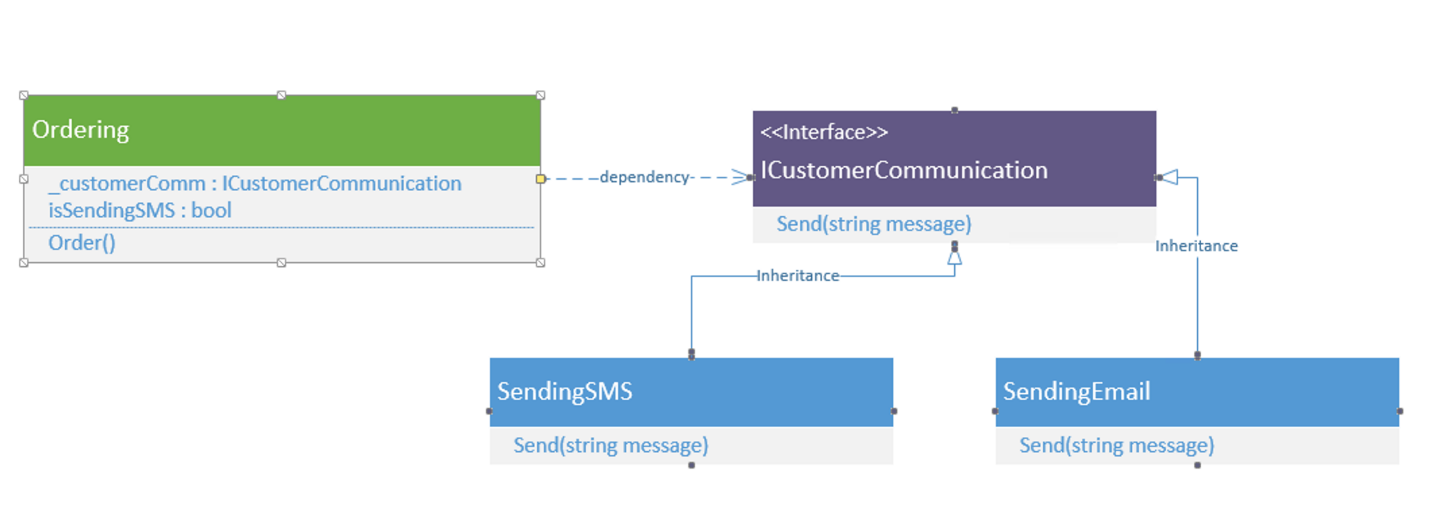


figure 6: UML Diagram

So what we have done here is that we have inverted the control to conform to the **DIP**. Now our high-level modules are dependent only on abstractions and not the lower level concrete implementations, which is exactly what **DIP** states.

# Dependency Injection (DI)

Although we already implement **IoC** in our example and the Ordering class depended on **ICustomerCommunication** abstraction. But we still use concrete classes in the **Ordering** class (a higher-level module). That prevents us from totally decoupling between classes.

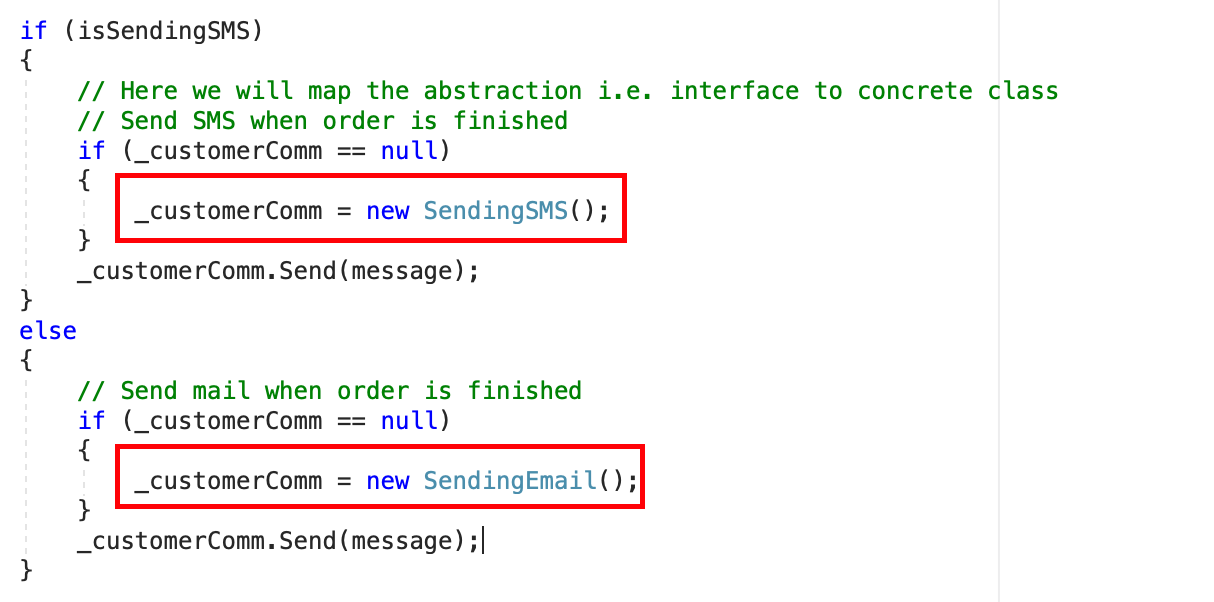


figure 7: concrete classes in Ordering class

That is why **DI** comes to the picture!

**DI** is mainly for injecting the concrete implementation into a class that is using abstraction (i.e. **ICustomerCommunication** interface). The main idea of **DI** is to reduce the coupling between classes and move the binding of abstraction and concrete implementation out of the dependent class. We can implement **DI** in 3 ways:

* **Constructor Injection**
* **Method Injection**
* **Property Injection**

1. **Constructor Injection**

With this approach, we will pass the object of the concrete class into the constructor of the dependent class and assign it to the interface is using.

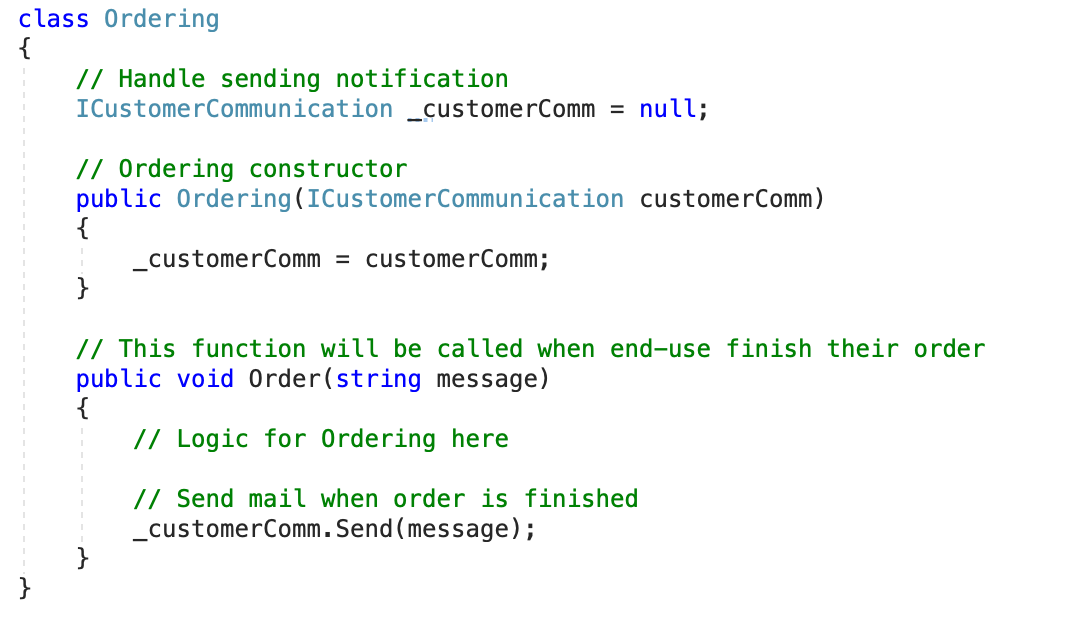


figure 8: Constructor Injection

In the above code, the constructor will take the concrete class object and bind it to the interface handle. If we need to pass the **SendingSMS**’s concrete implementation into this class, all we need to do is declaring an instance of **SendingSMS** class then pass it to **Ordering**’s constructor as below:

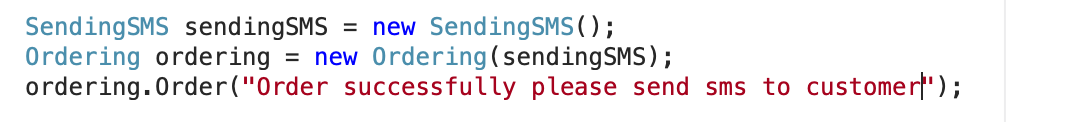


figure 9: using Constructor Injection

2. **Method Injection**

With using constructor injection we will have to use the instance of the concrete class — either **SendingSMS**or **SendingEmail**class during the lifetime of Ordering class. Now if we want to pass the instance of the concrete class on each invocation of the method we have to use **Method Injection** method.

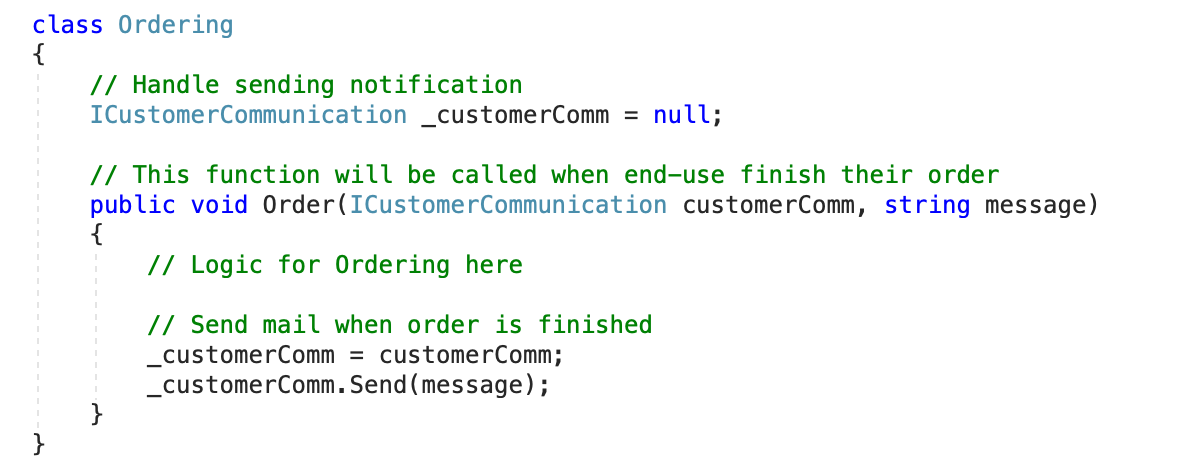


figure 10: Method Injection

And we will use **Method Injection**as below:

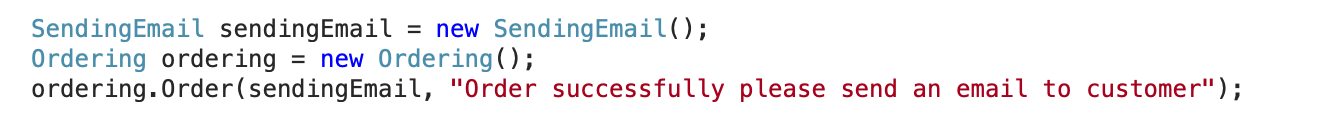


figure 11: Using Method Injection

3.**Property Injection**

Now we already know that the dependent class will use one concrete class for the entire lifetime with **Constructor Injection**approach and the **Method Injection** will impact to “method” level only. But what if the responsibility of the selection of concrete class and invocation of the method are in separate places. In such cases, we need property injection.

With this approach, we pass the object of the concrete class via a **setter property** that was exposed by the dependent class.

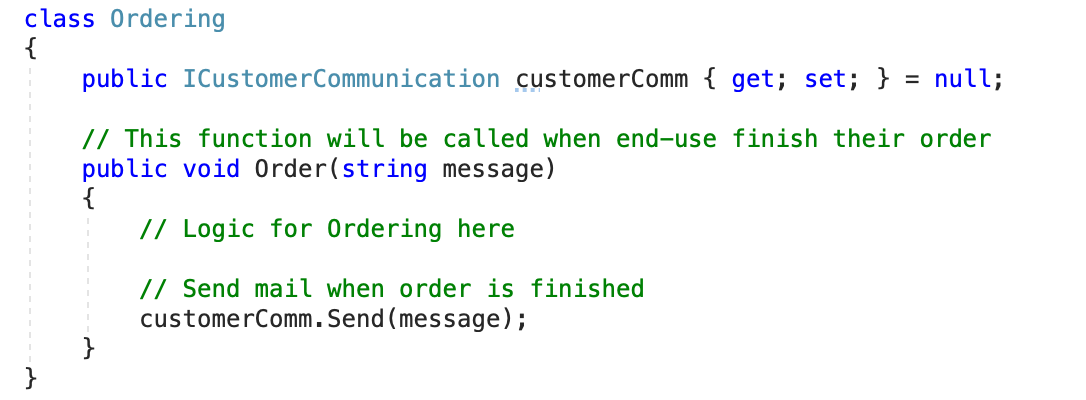


figure 12: Property Injection

And we will use “**Property Injection**” as below:

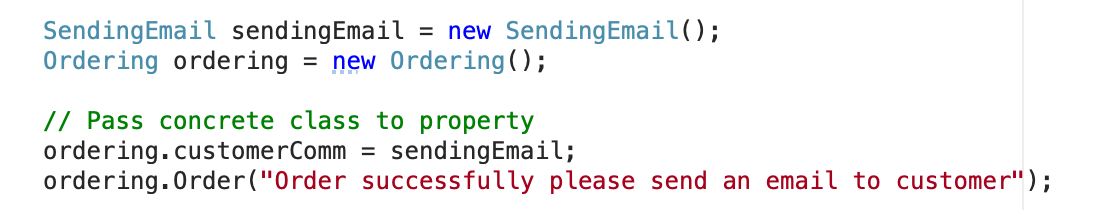


figure 13: User property injection

**Constructor Injection**is the most used approach when it comes to implementing the DI. If we need to pass different dependencies on every method call then we use **Method Injection**. **Property Injection** is used less frequently.

At this time, if you’re a beginner you can get understanding about **DIP**, **IoC** and **DI**. In the next section, we will implement **DI** with the built-in feature of .NET Core.

# Dependency Injection in ASP .NET Core

This is a built-in feature in ASP .Net Core. This supports is not limited to middleware, but also support in Controllers, Views, and Model as well. There are two types of service container provided by the ASP.net core: **Framework Services** and **Application Services**. The framework services are services that are provided by the ASP.net core such as **ILogger**. The **Application Services** are the custom services created base on our requirement.

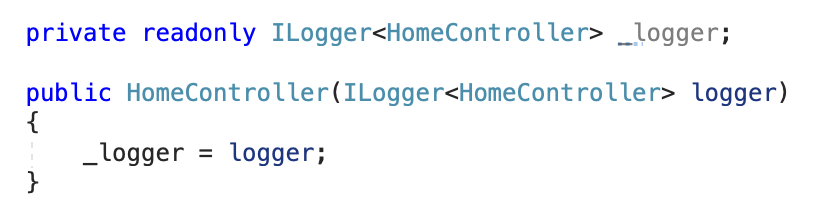


figure 14: Framework services — ILogger

## Dependency Injection in Controllers

ASP .Net Core has built-in support for constructor-based dependency. The dependency required by the controller is simply adding a service type to the controller in the constructor. The ASP .Net Core will identify the service type and try to resolve the type. Let’s make a sample!

Firstly, we will create a concrete named “**WelcomeMessage**” that inherited from “**IWelComeMessage**” interface — an abstraction.



figure 15: IWelcomeMessage interface and WelcomeMessage class

Now we need to add this service to the service container so that when the controller is requested for service, it is available to use. We can add the service to the service container in **ConfigureServices** method of startup class. There are three different life option available: **Transient**, **Scoped**, and **Singleton**(we will come back with these options later).

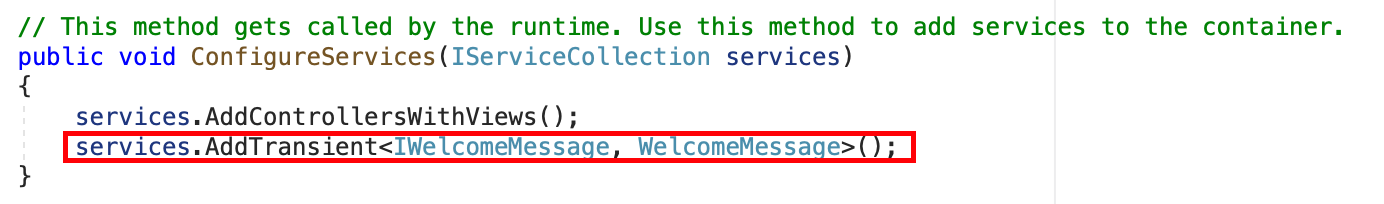


figure 16: Update ConfigureServices in Startup.cs

The last step, inject service into Controller via the constructor.



figure 17: DI in Controller

Here is the result:

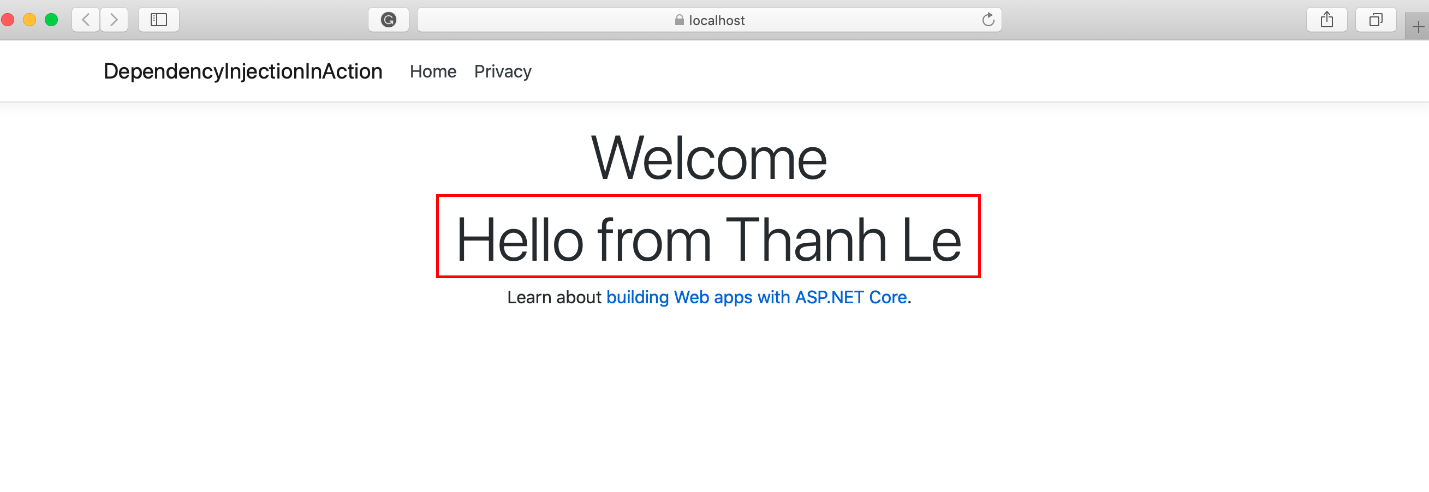


figure 18: DI result

Please make sure you register DI in **ConfigureServices** method of startup class, if not you will have the below error:

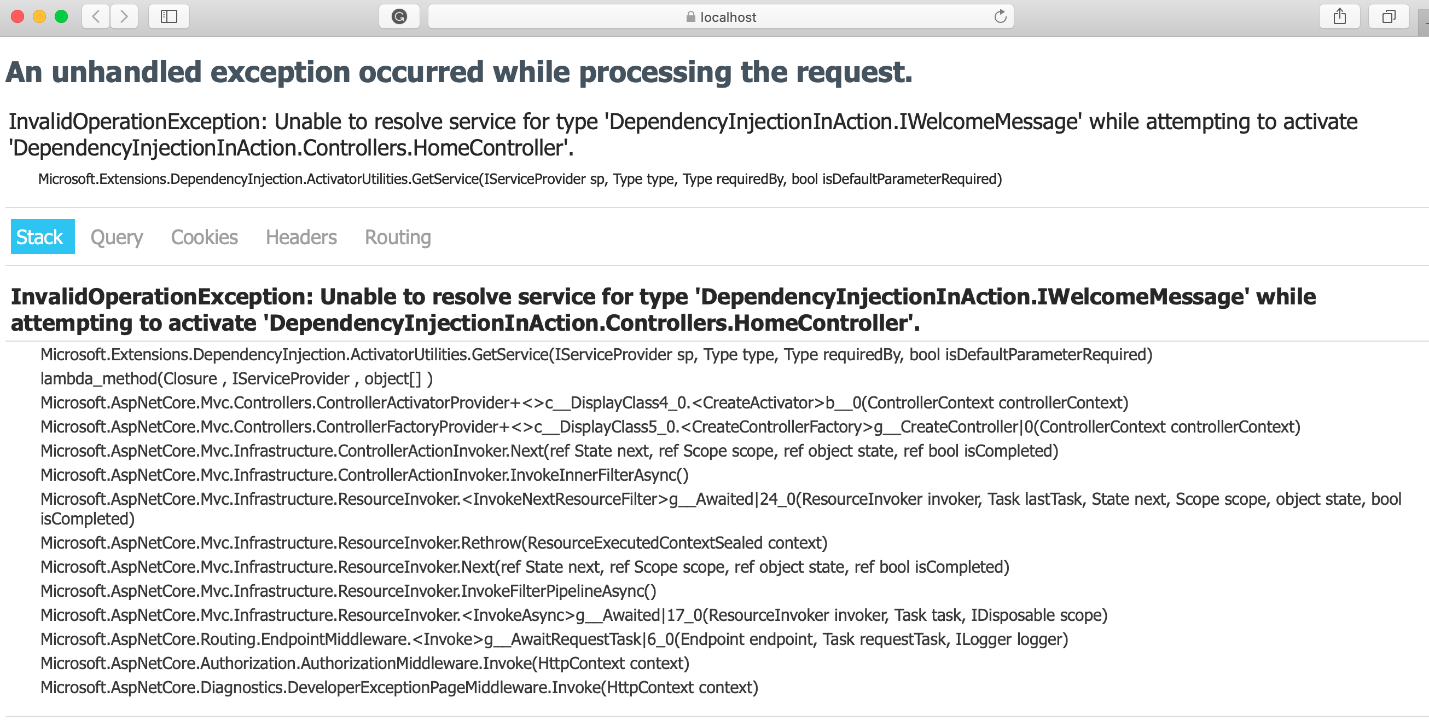


figure 19: DI error

## Inject the dependency in Controller methods/actions — Method Injection

ASP .Net Core allows us to inject the dependency into particular action using “**FromServices**” attribute. This attribute tells the ASP .Net Core framework that parameter should be retrieved from the service container.



figure 20: Method injection

And the result still the same

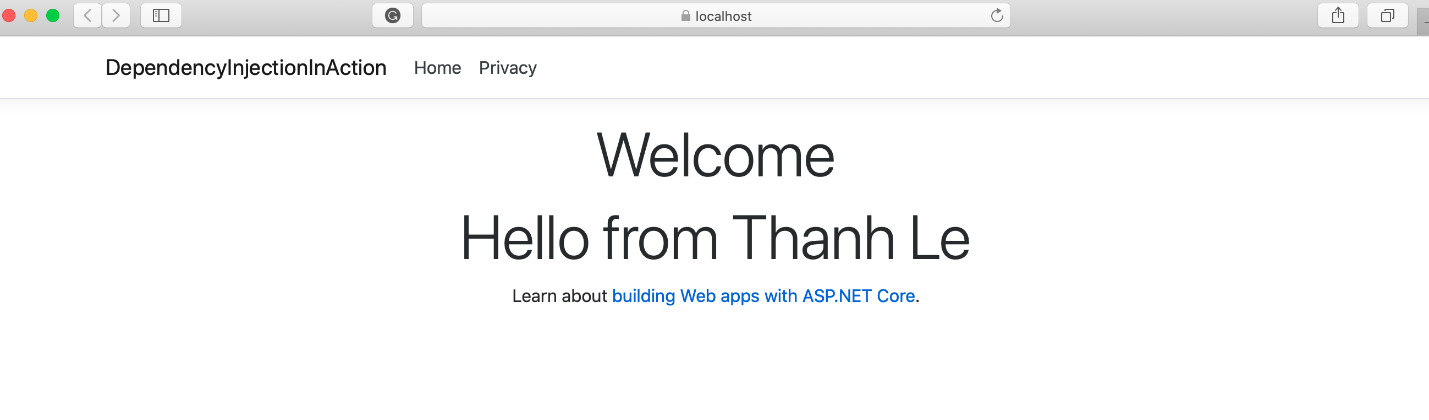


figure 21: Method Injection result

*Note: The property injection is not supported by the ASP .Net Core*

## Inject service manually

In this method, service is not injected in controller constructor or in the Controller action as a parameter. Using method “**GetService**” of “**HttpContext.RequestServices**” property, we can get dependent services configured with Service container.



figure 21: Inject service manually

## Inject service into Views

ASP .Net Core can also able to inject the dependency to View. This is very useful to inject service that related to View such as “localization”. We can inject the service dependency into the view using @inject directive.



figure 22: Inject service to View

**View Injection** can be used to populate the UI elements such as dropdown. The common dropdown-list such country dropdown can be populated from the service. Rendering such things from the service is the standard approach in ASP .Net Core. Alternatively, we can use ViewBag and ViewData to populate the dropdown. The directive **@inject** is also be used to override the injected service. For example, we are using Html helper service for the rendering the Html tags such as dropdown, textbox, etc. We can replace this service with our own service using **@inject** directive.

# Service Lifetime

ASP .Net Core allows us to specify the lifetime for registered services. The service instance gets disposed automatically based on specified life-time. So we do not care about the cleaning this dependency, it will take care by ASP .Net Core framework. There are 3 types of life-times.

## Singleton

The application will create and share a single instance of the service through the application life. The service can be added as a singleton using **AddSingleton** method of **IServiceCollection**. ASP .Net Core creates service instance at the time of registration and subsequence request use this service instance.

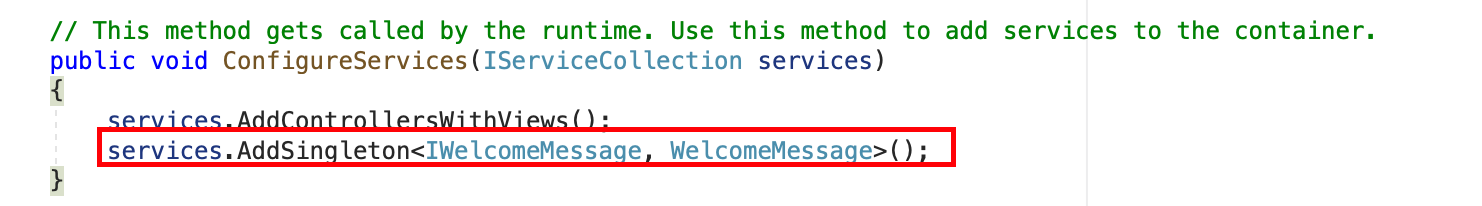


figure 23: Singleton lifetime

## Scoped

ASP .Net Core will create and share an instance of the service per request to the application. It means that a single instance of service available per request. It will create a new instance for every new request. The service can be added as scoped using **AddScoped** method of **IServiceCollection**in **ConfigureServices (Startup**class**)**.

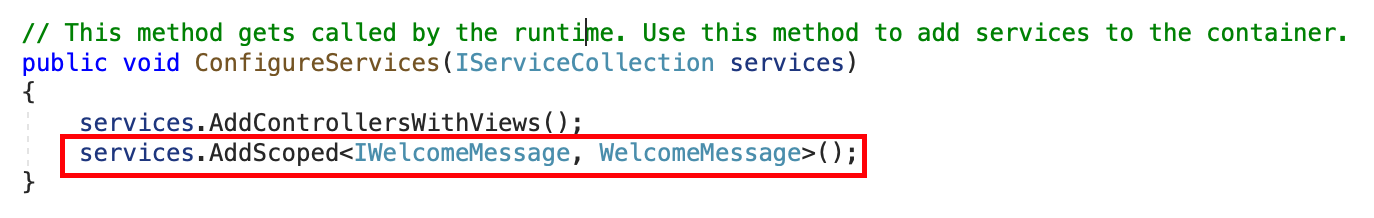


figure 24: Scoped lifetime

## Transient

ASP .Net Core will create and share an instance of the service every time to the application when we ask for it. The service can be added as **Transient** using **AddTransient** method of **IServiceCollection**. This life-time can be used in stateless service. It is a way to add lightweight service.

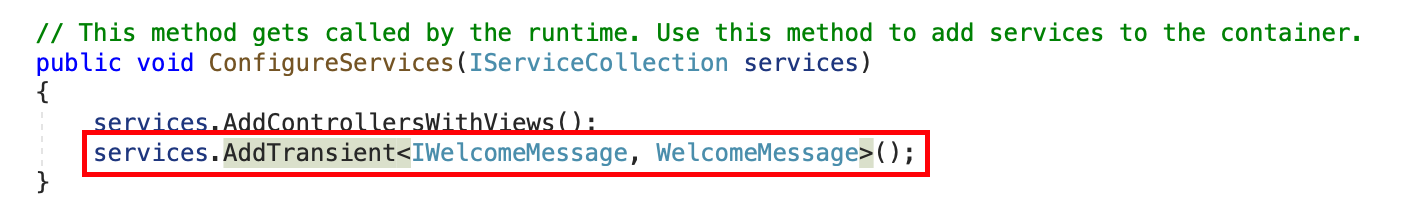


figure 25: Transient lifetime

# Conclusion

Dependency Injection (**DI**) is one of the most important design patterns in software development. This will help us to create a loosely coupled application so that it has provided greater flexibility, maintainability, testability and also reusability. With the built-in feature in ASP. NET Core we can easily apply **DI**to my applications.

# References

## [Dependency injection in ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1" \t "_blank)

### [By Steve Smith and Scott Addie ASP.NET Core supports the dependency injection (DI) software design pattern, which is a…](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1" \t "_blank)

#### [docs.microsoft.com](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1" \t "_blank)

## [596dip.pdf](https://drive.google.com/file/d/0BwhCYaYDn8EgMjdlMWIzNGUtZTQ0NC00ZjQ5LTkwYzQtZjRhMDRlNTQ3ZGMz/view" \t "_blank)

### [Dependency Inversion](https://drive.google.com/file/d/0BwhCYaYDn8EgMjdlMWIzNGUtZTQ0NC00ZjQ5LTkwYzQtZjRhMDRlNTQ3ZGMz/view" \t "_blank)

#### [edrive.google.com](https://drive.google.com/file/d/0BwhCYaYDn8EgMjdlMWIzNGUtZTQ0NC00ZjQ5LTkwYzQtZjRhMDRlNTQ3ZGMz/view" \t "_blank)

## [Dependency inversion principle](https://en.wikipedia.org/wiki/Dependency_inversion_principle" \t "_blank)

### [In object-oriented design, the dependency inversion principle is a specific form of decoupling software modules. When…](https://en.wikipedia.org/wiki/Dependency_inversion_principle" \t "_blank)

#### [en.wikipedia.org](https://en.wikipedia.org/wiki/Dependency_inversion_principle" \t "_blank)