# Using Dependency Injection and Configuration

In this lesson we will learn about the dependency injection technique and the .NET configuration system.

## What is dependency injection?

To understand what dependency injection is we can start by thinking about a class that uses some other class. For instance, our ItemsController uses the ItemsRepository. When you have this relationship where one class uses another class we say that this second class is a dependency of the first class.

Now, if instead of constructing the ItemsRepository object directly as we are doing so far, we receive an instance of ItemsRepository in the ItemsController constructor, then we are injecting ItemsRepository into ItemsController. This is what we call dependency injection. But also notice that the constructor is not exactly receiving an instance of ItemsRepository, but instead it receives an interface called IItemsRepository. This relates to what we call the Dependency Inversion Principle.

The Dependency Inversion Principle states that code should depend on abstractions as opposed to concrete implementations. Let's bring again our class and one of its dependencies, Dependency A. Currently, the class depends directly on Dependency A. However, we can change this so that the class instead depends on an abstraction, which in the case of C# is an interface, and then we can have the dependency depend on or implement the interface.

Why would we do this? Well, by doing this our class implementation is decoupled from the dependency implementation in such a way that if we ever decide to switch to a different or updated dependency our class does not need to change at all. The only thing that the dependencies need to do is to implement the interface that our class depends on.

So, by having our code depended upon abstractions we are decoupling implementations from each other. And, this also makes our code cleaner, easier to modify and easier to reuse.

## How to construct the dependencies?

Now that our code is decoupled from these dependencies, we have a new problem. How and who is going to construct the dependencies? Image once again that we have our class, which depends on an interface implemented by Dependency A. Since our class cannot construct the dependency directly, because it only knows about the interface, a third actor comes into play which is called the “Service Container”, which in ASP.NET Core is known as IServiceProvider.

During application startup, which typically happens in Startup.cs in ASP.NET Core apps, we would register our dependencies with the service container and this one in turn builds a map of all the registered dependencies. Then, eventually, when our class needs to get constructed, the service container first finds, constructs the dependency (or reuses it if it has already been constructed) and then injects it into our class as it creates our class instance.
This is very powerful because you could even have dependencies that depend on other dependencies but, as long as they all have been registered with the service container, they will all be resolved properly whenever any class needs them during the application lifecycle.

## ASP.NET Core Configuration

Let's also talk about configuration. At this point our service is able to talk to our MongoDB database via the host and port information that we have hard coded in our service implementation. However, this is not ideal since eventually the MongoDB docker container will live outside of our local box where localhost will likely not make sense and port 27017 is not guaranteed to be available.

Luckily ASP.NET Core supports a few configuration sources that are able to store and provide such configuration details to our service when needed. One of the most popular sources, especially for local development, is the appsettings.json file where we can store all sorts of configuration information that we expect could change across environments like the database host and port. And the good thing is that just like this one there are several other sources like command line arguments, environment variables, local secrets and even the cloud.

All these configuration sources are automatically read for us and loaded into the configuration system during host startup, which is configured in the Program.cs file in ASP.NET Core applications. And from there we can get access to them typically in our Startup.cs file.

Let's now update our Catalog service to take advantage of both dependency injection and the ASP.NET Core configuration system.