**ASP.NET Core dependency injection tutorial**

**HomeController**

public class HomeController : Controller  
{  
    private IEmployeeRepository \_employeeRepository;  
  
    // Inject IEmployeeRepository using Constructor Injection  
    public HomeController(IEmployeeRepository employeeRepository)  
    {  
        \_employeeRepository = employeeRepository;  
    }  
  
    // Retrieve employee name and return  
    public string Index()  
    {  
        return \_employeeRepository.GetEmployee(1).Name;  
    }  
}

**Please note :**

* HomeController is dependant on IEmployeeRepository for retrieving Employee data.
* Instead of the HomeController creating a new instance of an implement ion of IEmployeeRepository, we are injecting IEmployeeRepository instance into the HomeController using the constructor.
* This is called **constructor injection**, as we are using the constructor to inject the dependency.
* Notice, we are assigning the injected dependency to a **read-only field**. This is a good practice as it prevents accidentally assigning another value to it inside a method.
* At this point, if we run the project we get the following error  
  InvalidOperationException: Unable to resolve service for type 'EmployeeManagement.Models.IEmployeeRepository' while attempting to activate 'EmployeeManagement.Controllers.HomeController'.
* This is because the ASP .NET dependency injection container does not know which object instance to provide if someone requests an object that implements IEmployeeRepository
* IEmployeeRepository may have several implementations. At the moment in our project we only have one implementation and that is MockEmployeeRepository
* As the name implies, MockEmployeeRepository works with the in-memory employee mock data.
* Later,we will discuss providing another implementation for IEmployeeRepository which retrieves employee data from SQL Server database.
* For now, let's work with MockEmployeeRepository.
* To fix the InvalidOperationException error, we need to register MockEmployeeRepository class with the dependency injection container in ASP.NET core.
* We do this in ConfigureServices() method in Startup class

**Registering Services with the ASP.NET Core Dependency Injection Container :**   
  
ASP.NET core provides the following 3 methods to register services with the dependency injection container. The method that we use determines the**lifetime of the registered service**.  
  
**AddSingleton()** - As the name implies, AddSingleton() method creates a **Singleton service**. A Singleton service is created when it is first requested. This same instance is then used by all the subsequent requests. So in general, a Singleton service is created only one time per application and that single instance is used throughout the application life time.  
  
**AddTransient()** - This method creates a **Transient service**. A new instance of a Transient service is created each time it is requested.   
  
**AddScoped()** - This method creates a **Scoped service**. A new instance of a Scoped service is created once per request within the scope. For example, in a web application it creates 1 instance per each http request but uses the same instance in the other calls within that same web request.  
  
Please do not worry, if this is a bit confusing at the moment. We will be revisiting these 3 methods several times in our upcoming concepts.  
  
For now, to fix the InvalidOperationException error, let's register MockEmployeeRepository class with the ASP.NET Core Dependency Injection container using AddSingleton() method as shown below. So, with this code in-place, if someone asks for IEmployeeRepository, an instance of MockEmployeeRepository will be provided.

public void ConfigureServices(IServiceCollection services)  
{  
    services.AddMvc();  
    services.AddSingleton<IEmployeeRepository, MockEmployeeRepository>();  
}

At this point, you might be thinking, why do we have to do all this. Why can't we simply create an instance of MockEmployeeRepository class in the HomeController using the **new**keyword as shown below.

public class HomeController : Controller  
{  
    private readonly IEmployeeRepository \_employeeRepository;  
  
    // Inject IEmployeeRepository using Constructor Injection  
    public HomeController(IEmployeeRepository employeeRepository)  
    {  
        \_employeeRepository = new MockEmployeeRepository();  
    }  
  
    // Retrieve employee name and return  
    public string Index()  
    {  
        return \_employeeRepository.GetEmployee(1).Name;  
    }  
}

Well, this makes HomeController tightly coupled to MockEmployeeRepository. Later if we provide a new implementation for IEmployeeRepository and if we want to use that new implementation instead of MockEmployeeRepository, we have to change the code in HomeController. You might be thinking, this is just one line of code change, so it is not that difficult to do.  
  
Well, what if we have used this MockEmployeeRepository in 50 other controllers in our application?  
The code in all the 50 controllers has to change. This is not only tedious but also error prone.  
  
So in-short, using **the**new keyword to create instances of dependencies creates tight coupling and as a result your application will be difficult to change. With dependency injection we will not have this tight coupling.   
  
With dependency injection, even, if we have used MockEmployeeRepository in 50 other controllers in our application, if we want to swap it out with a different implementation, we just need to change the following one line of code in Startup.cs file. Notice, we are now using DatabaseEmployeeRepository instead of MockEmployeeRepository.

public void ConfigureServices(IServiceCollection services)  
{  
    services.AddMvc();  
    services.AddSingleton<IEmployeeRepository, DatabaseEmployeeRepository>();  
}

Unit testing also becomes much easier, as we can easily swap out dependencies with dependency injection.  
  
Don't worry, if this is slightly confusing. We will provide a different implementation for IEmployeeRepository in our upcoming videos. This new implementation will retrieve data from a SQL Server database. We will then replace the MockEmployeeRepository implementation with the DatabaseEmployeeRepository implementation. At that point, you will understand the power and flexibility dependency injection provides.