Dependency injection for unit test in ASP.NET CORE Web Application

Introduction

The most challenging thing I found during my learning process is to apply a unit test to my .Net Core website project. When I heard about dependency injection, my mind forced me not to deal with it; why? , it’s because of the name and the concept behind it. But on the other hand, I need to make sure my application works fine during the deployment stage. To do that, I decided to write a few unit tests on my project. Since I use ORM to the data source, applying unit tests becomes harder. So at this time, I decided to face my fear and find a solution.

Student record

How to use it in the .NET framework

Dependency Service lifetime

Implementing Dependency injection into controllers

Dependency in a unit test.

What is dependency injection?

It is a software design pattern that enables the development of loosely coupled code.

What does it mean by loosely coupled code? In short, loosely coupled denotes a low degree of dependence on software components on each other, which makes two objects less independent. This type of design approach reduces the risk when one element is changed.

In addition, dependencies should be on interfaces because the interface allows the injector to tell the client which variables and methods are available during calls and provide necessary services. This technique enables the code to be extensible and easily testable.

The benefit of using dependency injection

* Make the application more testable.
* Decouple the application
* Control over a lifetime of dependencies
* Provide code reusability and maintainability.

Type of Dependency injections.

There are three types of DIs.

1. Constructor Injection :
   1. It is a process of using a constructor to pass the dependencies class.
   2. Preferred to use if the class cannot work without a dependency
2. Property Injection
   1. The injection takes place using property
   2. Dependencies can also be injected by declaring an internal or public property.
3. MethodInjection (Parameter-based Injection)
   1. Simple to inject by creating methods that accept dependency as an argument.
   2. They are preferred to use if the dependency needs to be renewed after each use.

So far, you have an idea of what/why dependency injection is helpful in asp.net. Below you will find a sample project to show how easy and beneficial to use this design pattern/technique.

In this project, you will find all you need to know about how dependency injection work and how to use it in the unit test; I choose the MVC .NET Core project with an entity framework to record student data into the database. Before we begin, I expect you to have at least a basic knowledge of the Entity framework. If not, please check out this link before <https://docs.microsoft.com/en-us/ef/core/get-started/overview/first-app?tabs=netcore-cli>

NuGetPackage used in this project.

1. Microsoft.EntityFrameworkCore.SqlServer
2. Microsoft.EntityFrameworkCore.tools
3. Microsoft.AspNetCore.Diagonstic.EntityFramework

And of course, you can find the complete project here:-

Let’s begin.

First, create a new .NET core MVC project, and rename it StudentRecord. For this example, I am using the Entity framework and local database.

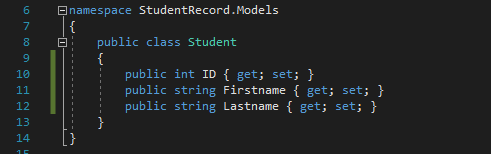
We will create one database model to store the student’s id, first name, and last name.

Since we use EntityFramework, we follow a code-first approach to creating a database. If you are not familiar with the code-first process, please do visit the link below.

[A Simple Step to Entity Framework: Code First Approach](https://www.c-sharpcorner.com/UploadFile/a9d961/a-simple-step-to-entity-framework-code-first-approach/)

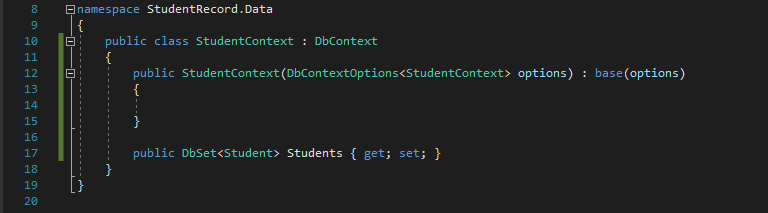
The most straightforward way to apply a code-first approach is simply needed to create a new Student.cs file inside the model folder and add student class in the database context class.

//create student class with a property of id, first name, and last name.



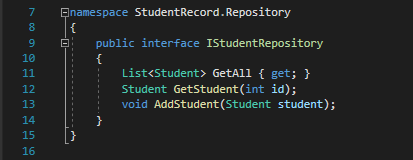
Now let’s create a database context class. So create StudentContext.cs inside Data folder

//add student model in to StudentContext class

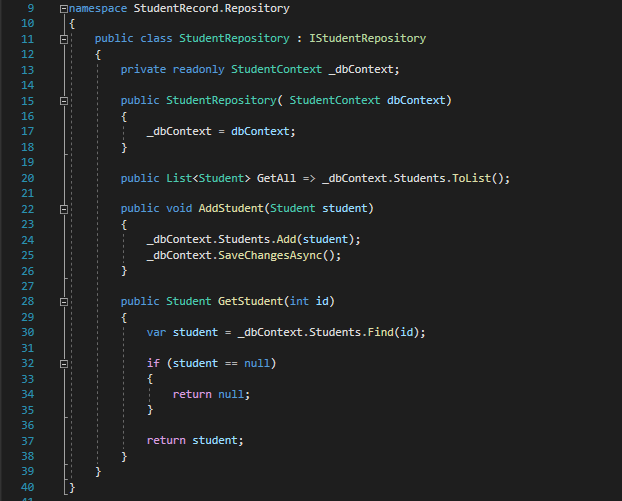


Now let’s create StudentRepository.cs class to do all the logic and IStudentRepository.cs interface to apply dependency injection later. Add both files into the Repository folder.

//IStudentRepository.cs

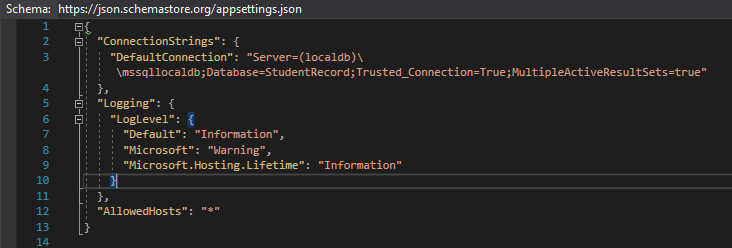


//StudentRepository.cs

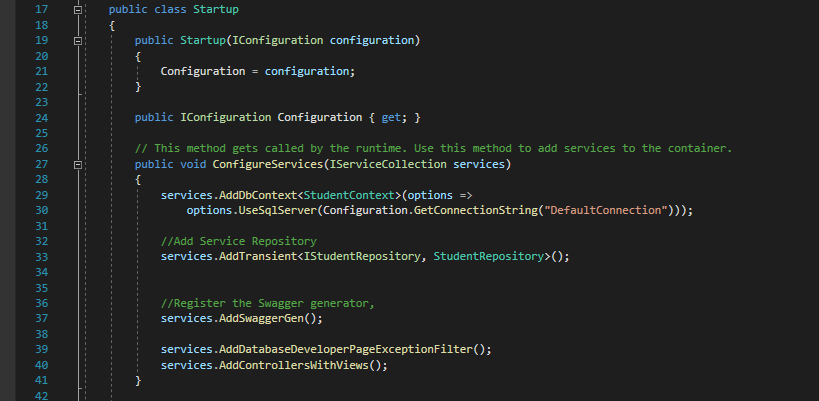


Now let’s add the interface into the Startup class. Doing this step enables the application to handle a dependency lifetime; first, let’s add the database connection string into the appsetting.json file and database context.

//add connection string into appsetting.json file



//configure startup file, inside ConfigureService method

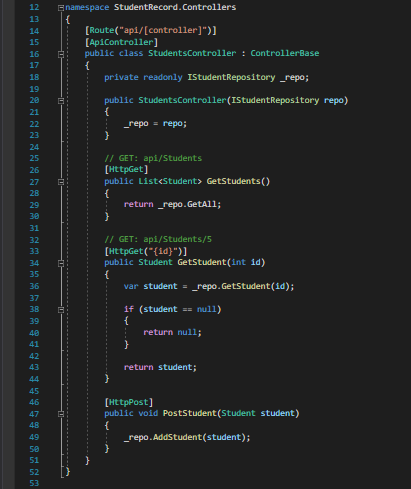


If you take a look closer, I use AddTransient() Service lifetime to register dependency. The reason for that is because we are using an effortless application with minimal requests. This type of lifetime is created in each request and disposed of it at the end request. There are two more Service LifeTimes in ASP.NET. Check out the link for the detailed [Dependency injection in .NET](https://docs.microsoft.com/en-us/dotnet/core/extensions/dependency-injection#service-lifetimes)

The next step will be creating an API controller and apply the constructor injection principle. Add StudentsController.cs file inside the controllers folder.

Detailed dependency

//StudentsController.cs



By now, you have a working API that handles a few requests. The API we created can list all students, display students by id, and even make new student records in the database. Besides, all the methods depend on the IStudentRepsoitory interface; we choose to use constructor dependency injection because of that.

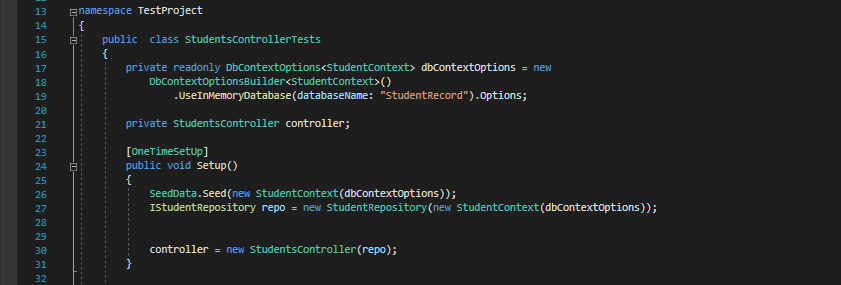
Writing tests for your project will help you get confidence in your product and save you a lot of pain and time. So far, we create a .net core web application, but we didn’t know if the code works appropriately or not unless we run and check manually. As I mentioned from the beginning, we will add a new project to run a simple unit test using dependency injection.

First, add a new NUnit project and download the following NuGet Package.

1. Microsoft.AspNetCore.Mvc.Core
2. Microsoft.EntityFrameworkCore.InMemory

To make it simple, let’s add the StudentControllerTests.cs class to test the API controller. Inside the test class, configure database context using UseInMemoryDatabase() and configure/Create StudentsController class to access all methods.

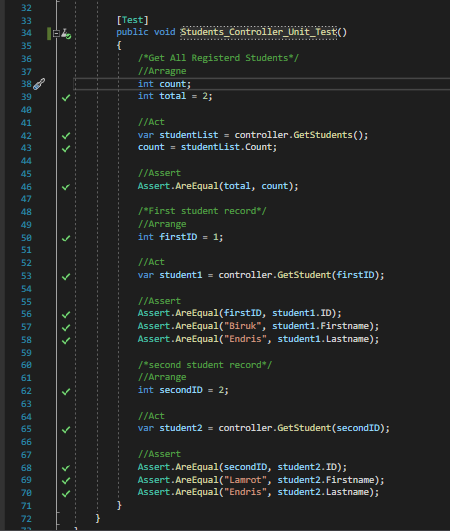
//Configure DB context option and use a controller

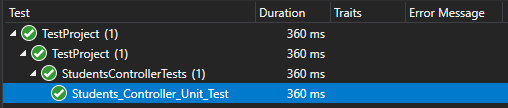


If you take a look, we have SeedData class to seed the database.



Finally, write a Unit Test and run.





.

Conclusion

Dependency injection is a powerful, sound, and critical technique to use in order to write clean, loosely coupled, easy to maintain code. There are three ways to do dependency injection, each having its own use case. Learn when to use these three techniques, and you will be well on your way to writing excellent, testable, and lovely code.

Reference source

1. <https://www.yogihosting.com/aspnet-core-dependency-injection/>
2. <https://docs.microsoft.com/en-us/dotnet/core/extensions/dependency-injection>
3. <https://entityframeworkcore.com/articles/carloscds-ef-core-dependency-injection>
4. <https://stackify.com/dependency-injection-c-sharp/>
5. <https://medium.com/flawless-app-stories/dependency-injection-and-its-type-e1c8d5da46f3>