**Tạo WEB API với kiến trúc Repository và Dependency Injection**

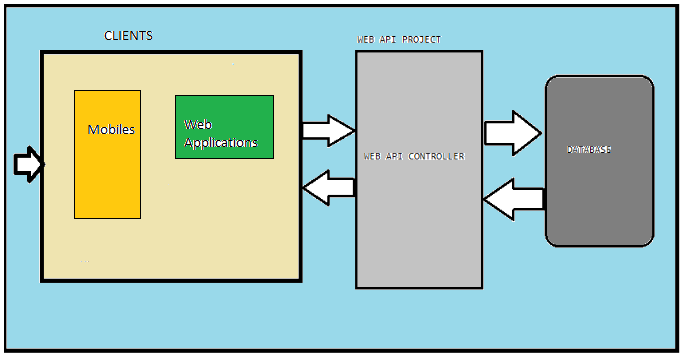
**Nội dung cần biết:**

* Repository Pattern là gì?
* Generic Repository Pattern là gì?
* Dependency Injection là gì?

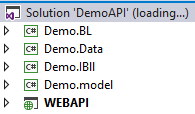
**Nội dung trong bài học:**

* Các lớp kiến trúc (Layered Architecture) trong Web API Project
* Làm việc với Repository Pattern trong Web API
* Làm việc với Dependency Injection trong Web API

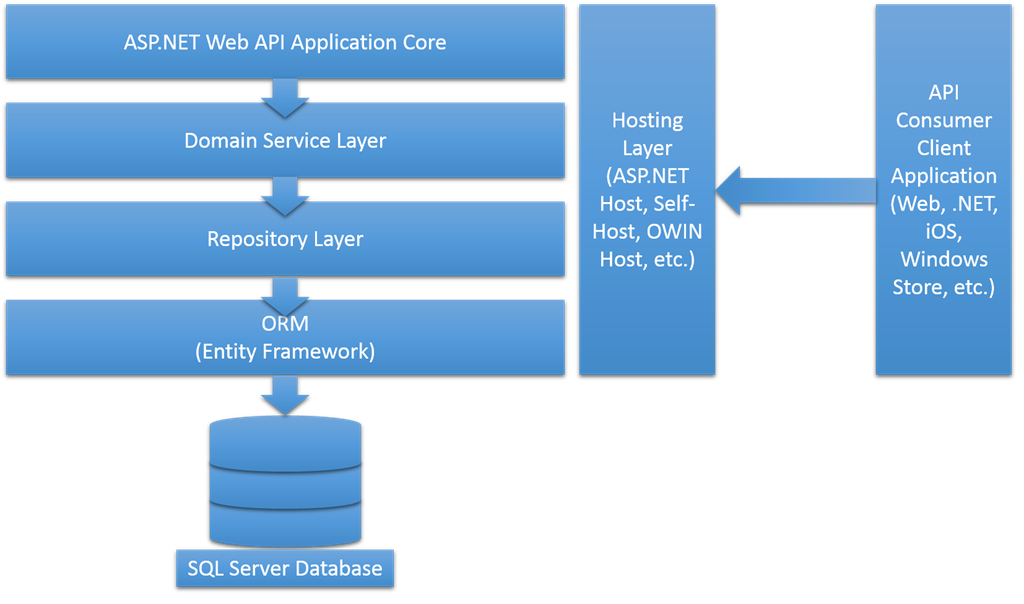
Thông thường, Web API chỉ được đóng gói bussiness logic, được sử dụng bởi các client khác nhau để tạo ra các ứng dụng client side như các ứng dụng di động hoặc ứng dụng Web:



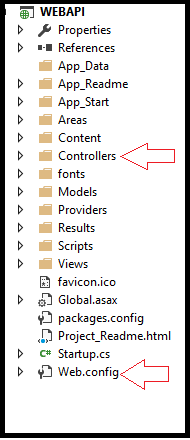
Bây giờ, chúng ta sẽ tạo ra một Web API với nhiều tầng, như sau:



Đây là hình ảnh hoàn chỉnh về cách làm việc giữa các lớp sau khi chúng ta lưu trữ (hosting):



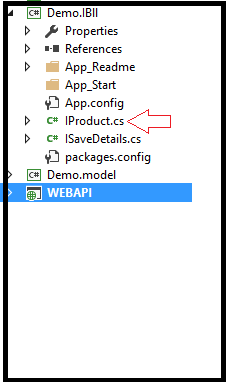
Bây giờ chúng ta sẽ kiểm tra chi tiết từng lớp, đầu tiên là lớp **Web API**:



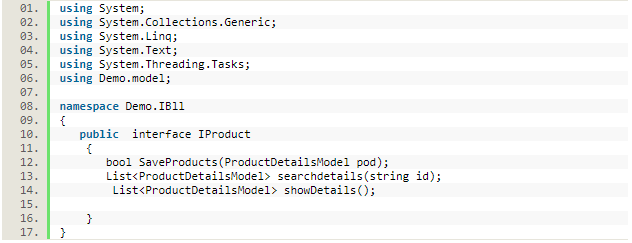
Lớp này chủ yếu xử lý các yêu cầu đến từ phía Client, chứa các Controller Classes. Các controller classes này được kế thừa từ API Controller. Trong mỗi Controller đều có các Action Methods và chúng ta sẽ viết chi tiết code trong mỗi Method đó.

**Lớp thứ 2 là lớp IBLL**, lớp này chứa các interface, nơi mà tất cả các abstract classes có phương thức abstract được khai báo. Vì vậy Controller sẽ gọi tới lớp này, nơi mà các interface của chúng ta đã được định nghĩa.

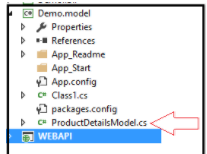
Trong lớp này, ta sẽ tạo một interface Iproduct.cs, chúng ta sẽ khai báo tất cả các methods abstract (phương thức trừu tượng) để thao tác với một số hoạt động:



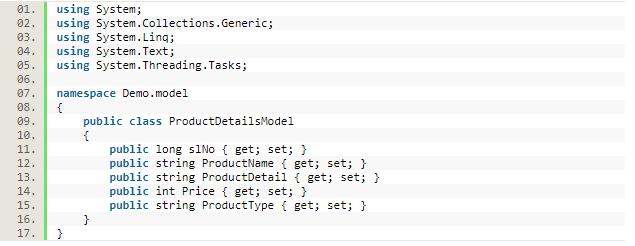
Ta khai báo 3 phương thức abstract tại **Iproduct.cs** như sau:



**Lớp Models:** nơi sẽ chứa tất cả các model classes và Properties được khai báo:

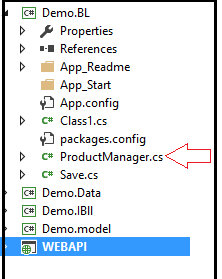


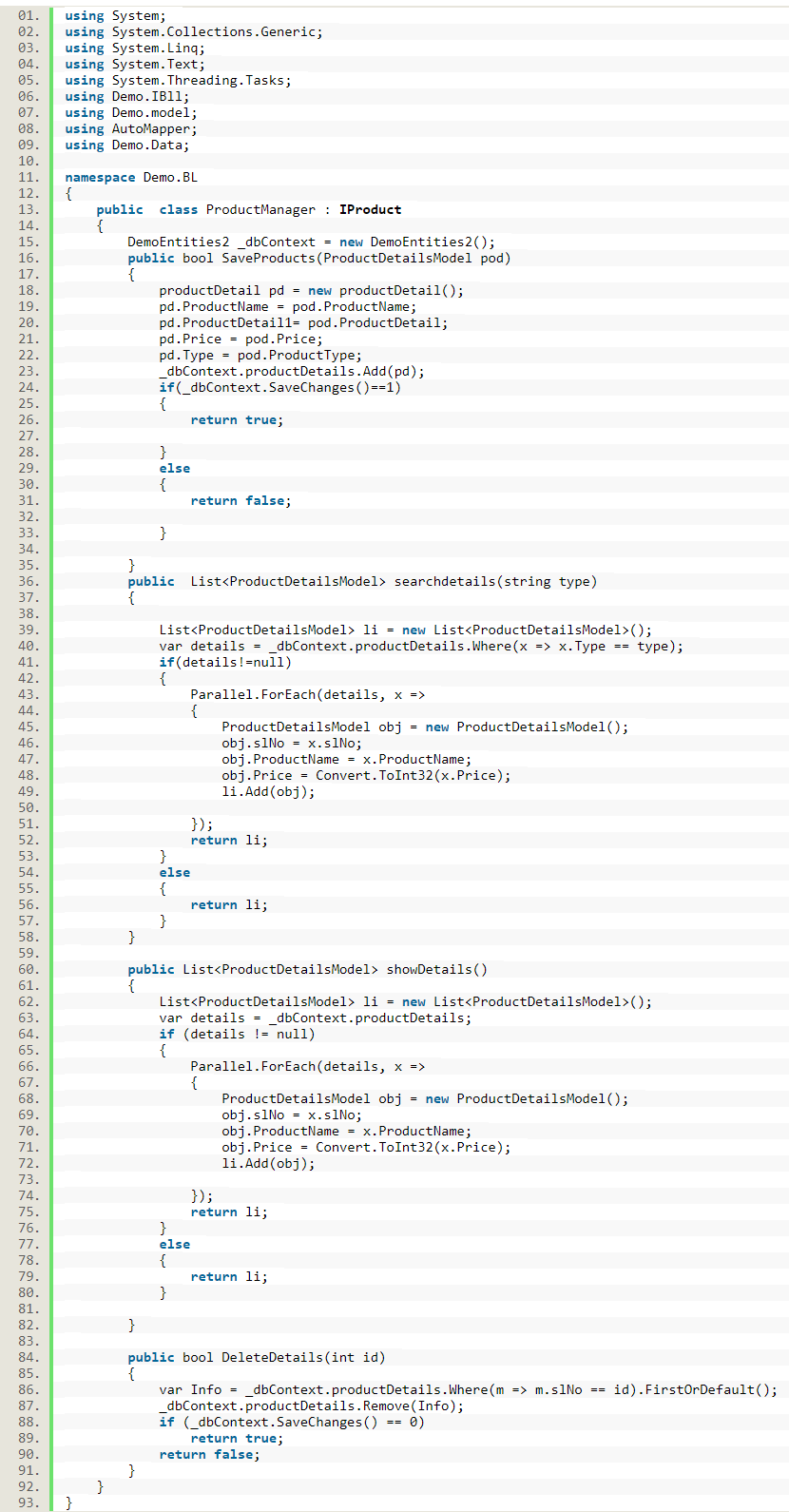
Trong lớp này ta sẽ tạo 1 **ProductDetailsModel.cs** với nội dung như sau:



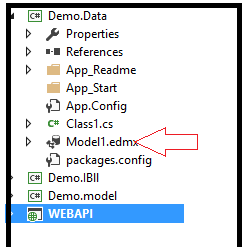
**Lớp BL:**

Tiếp theo chúng ta sẽ khai báo một Helper Class **ProductManager.cs**, kế thừa interface Iproduct vừa tạo ở trên:





Lớp Data: lớp này sẽ thực hiện công việc thao tác với dữ liệu (database)



Tại đây, ta tại một Model1.edmx như sau. Tầng này sẽ chịu trách nhiệm liên lạc với cơ sở dữ liệu và chịu tất các trách nhiệm liên quan tới DB. Sử dụng EntityFramework, sử dụng ORM Patterm.

*ORM là gì? – là một kỹ thuật, cơ chế lập trình thực hiện ánh xạ cơ sở dữ liệu sang các đối tượng trong các ngôn ngữ lập trình như Java, C#...*

**Sử dụng Dependency injection trong Web API:**

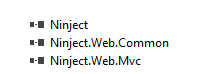
DI là gì? – DI là một kỹ thuật sử dụng để phát triển ứng dụng một cách độc lập. Độc lập nghĩa là một mô-đun của ứng dụng phải là duy nhất và không bị phụ thuộc và các mô-đun khác.

*Tight Coupling*: nghĩa là 2 lớp hoặc 2 mô-đun bị phụ thuộc lẫn nhau. Mọi thay đổi của một lớp đối tượng có thể gây ra ảnh hưởng tới một số thành phần của lớp hoặc mô-đun kia. Khi chúng ta tạo ra một object của một class và gọi tới một phương thức của class đó bởi một object từ class khác. 2 lớp này sẽ được coi là chặt chẽ hoặc phụ thuộc lẫn nhau.

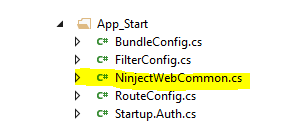
*Losse Coupling*: nghĩa là 2 đối tượng độc lập với một đối tượng có thể sử dụng một đối tượng khác mà không bị phụ thuộc vào nó. Chúng là duy nhất.

Sử dụng NinJect Container và Dependency Injection:

Trong Web Api Project, mở Manager Nuget Package và thực hiện cài đặt NinJect Container. Sau khi cài đặt xong ta sẽ có 3 assemblies trong References và 1 NinjectWebCommon.cs class trong App.start:



**NinjectWebCommon.cs** class in App.start



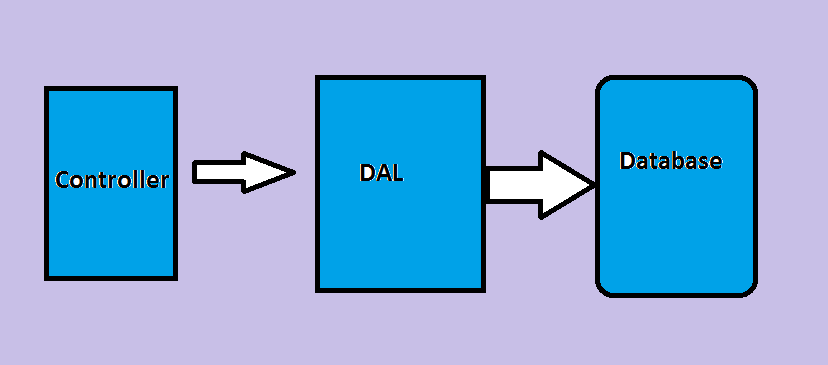
Mở class NinjectWebCommon.cs và thực hiện đăng ký một dependency như sau:



**Repository Patterm**

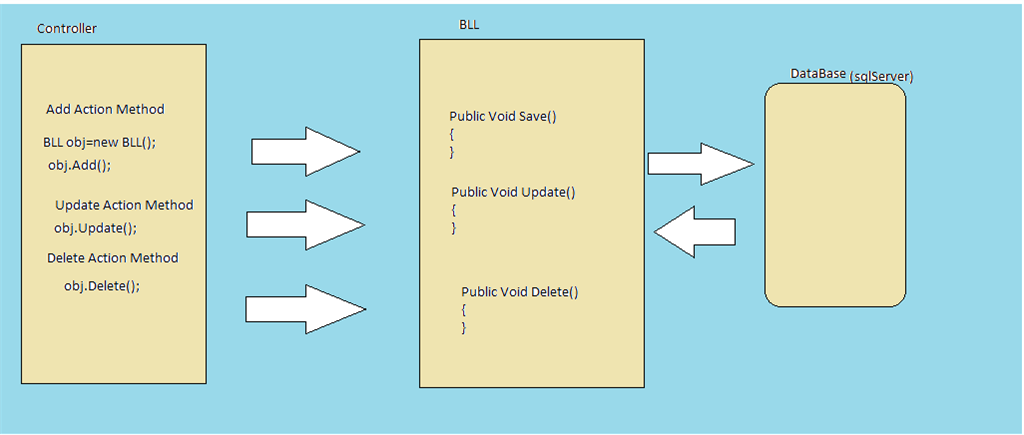
Khi chúng ta làm việc trong một khối kiến trúc, các project của chúng ta phải độc lập giữa tất cả các lớp. Hay nói cách khác lớp Controller và DataAccess phải độc lập nhau. Nếu chúng ta để chúng kết hợp chặt chẽ thì mọi thay đổi ở lớp DataAccess sẽ làm thay đổi mã Controller.

Nếu không có Repository thì cấu trúc ứng dụng của ta được thể hiện như hình:



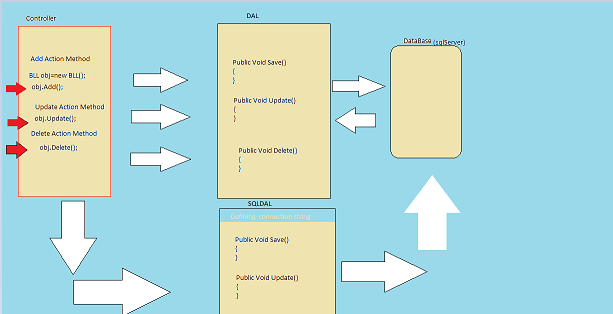
Do vậy, **Reposotory** được sinh ra giữa tầng **Data Access** và tầng **Bussiness** (nghiệp vụ) của ứng dụng. Mô hình này giúp bảo vệ ứng dụng khỏi những thay đổi từ kho dữ liệu và tạo điều kiện thuận lợi cho việc kiểm nghiệm.

Ở đây chúng ta đã tạo ra một đối tượng BLL trong controller., do vậy BLL và Controller đã được gắn kết chặt chẽ với nha => bất cứ thay đổi nào trong DAL đều sẽ ảnh hưởng tới toàn bộ Controller.



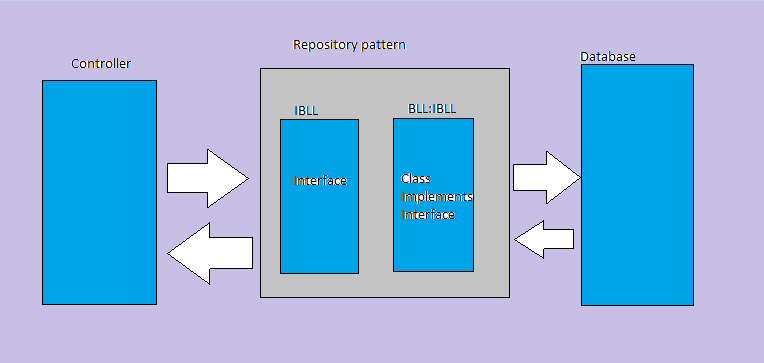
Giả sử chúng ta muốn thay đổi ADO.NET thay cho Entity Framework để thay đổi hiệu năng.

Do ứng dụng đang có sự gắn kết chặt chẽ nên khi ta muốn thay đổi là rất khó. Giả sử ta thay đổi tầng DAL thì tất cả các tầng controller sẽ cần phải thay đổi như sau:



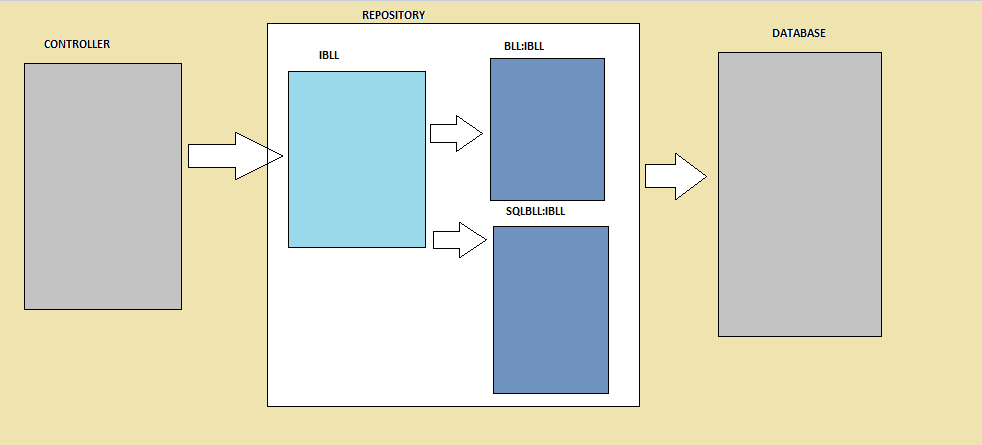
* Do vậy nó đã ảnh hưởng tới cả 2 tầng, như hỉnh ảnh thì tôi đã đánh dấu bằng mũi tên màu đỏ, là những nơi các ảnh hưởng có thể xảy ra. Khi chúng ta đang sử dụng đối tượng của lớp DAL thì nó cần phải được thay thế bằng tầng SQL DAL.

Bây giờ chúng ta sẽ sử dụng kiến trúc **Repository Pattern**:



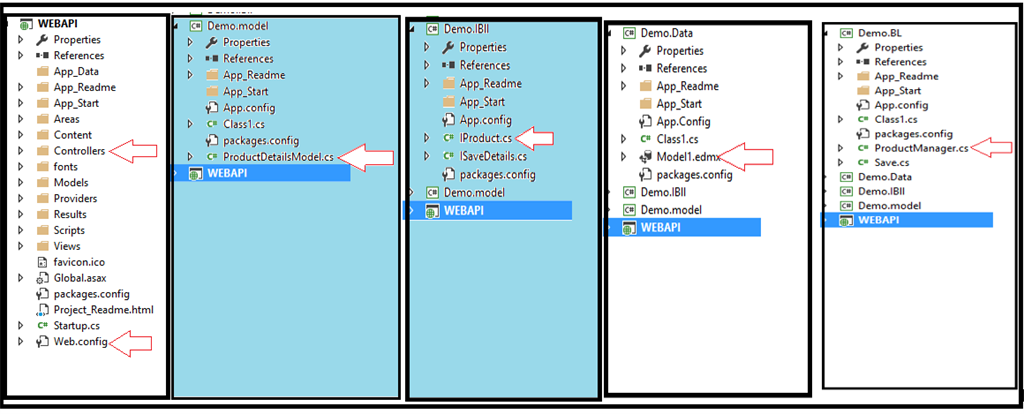
* ở đây các controller đã được trỏ tới IBLL, nó là các interface và các interface này được implimented trong BLL.

Giả sử bây giờ chúng ta muốn thay đổi Entity Framework BAL sang ADO.NET BAL thông thường. Chúng ta chỉ cần tạo ra một ADO.NET BAL và **kế thừa** nó từ **IBLL** như hình:



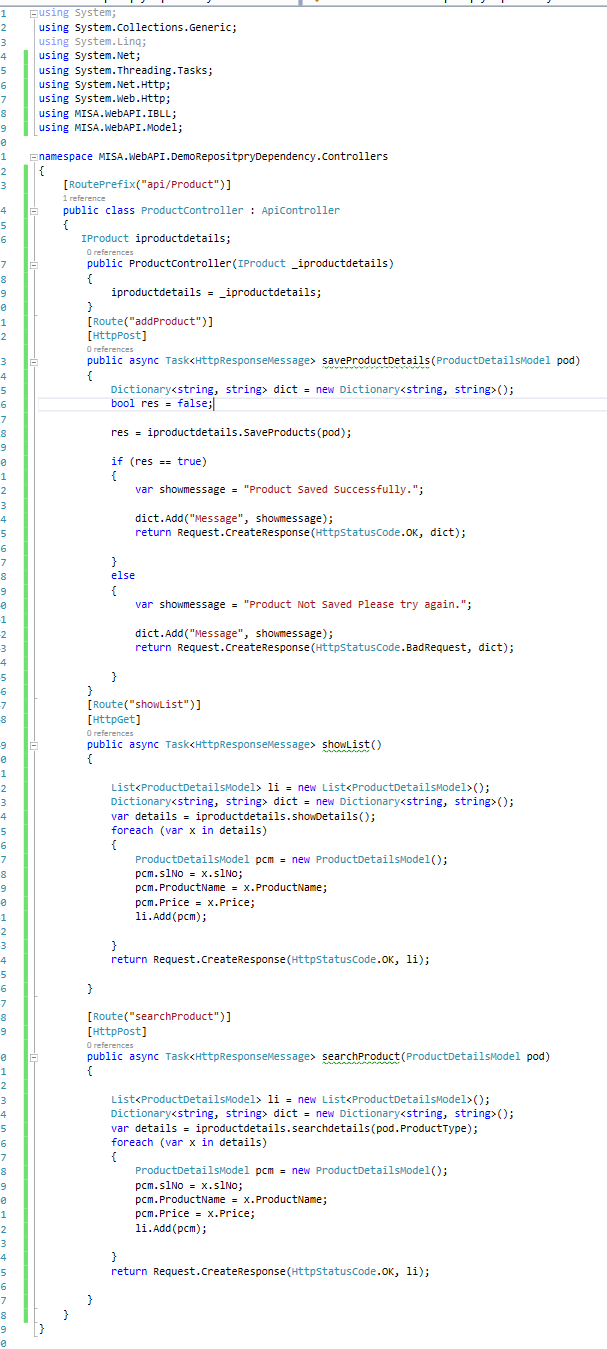
* Bằng cách này chúng ta có thể khắc phục được các vấn đề khi thực hiện thay đổi việc thay đổi bất cứ công nghệ truy cập dữ liệu .

**Vì lý do này nên chúng ta đã sử dụng IBLL và BLL trong dự án như hình:**



Giờ thì dự án của chúng ta gần như đã hoàn thành, ta cần viết thêm các mã trong controller cho bộ Dependency Injection và gọi các lớp tương ứng. Và cuối cùng mã hoàn chỉnh của chúng ta cho hoạt động CRUD trong ProductDetail controller sẽ như sau:

* Sau khi thực hiện xong Build Project và sử dụng Postman để kiểm tra kết quả.



Lợi ích của Repository:

1. Tập trung hóa được các logic về xử lý dữ liệu hoặc business logic.
2. Dễ dàng unit test
3. Đưa ra kiến trúc linh hoạt
4. Khi thay đổi logic của tầng data hoặc business logic, không cần thay đổi Repository
5. Giảm thiểu sự trùng lặp code
6. Đảm bảo các coder dùng chung 1 pattern
7. Ít lỗi hơn
8. Dễ dàng bảo trì sau này