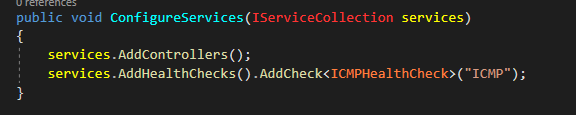
# Các services

1. Healthy check

Đầu tiên chúng ta tiến hành tạo implementation của **IHealthCheck ,** khi implement interface này, ta sẽ phải thục thi **CheckHealthAsync** và trong đó ta sẽ thực hiện những gì ta muốn để kiểm tra một cái gì đó.



Tạo service cho tác vụ check này với tên mà ta muốn



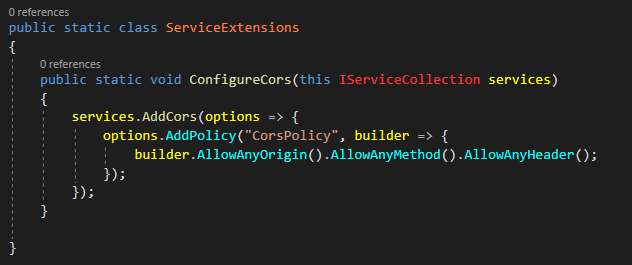
Và cuối cùng, để kích hoạt service này chạy ta sẽ add một route tới Config

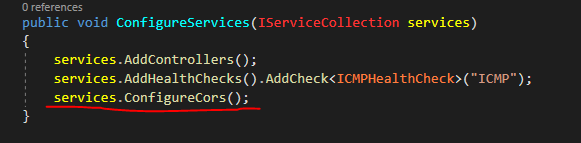


1. CORS config

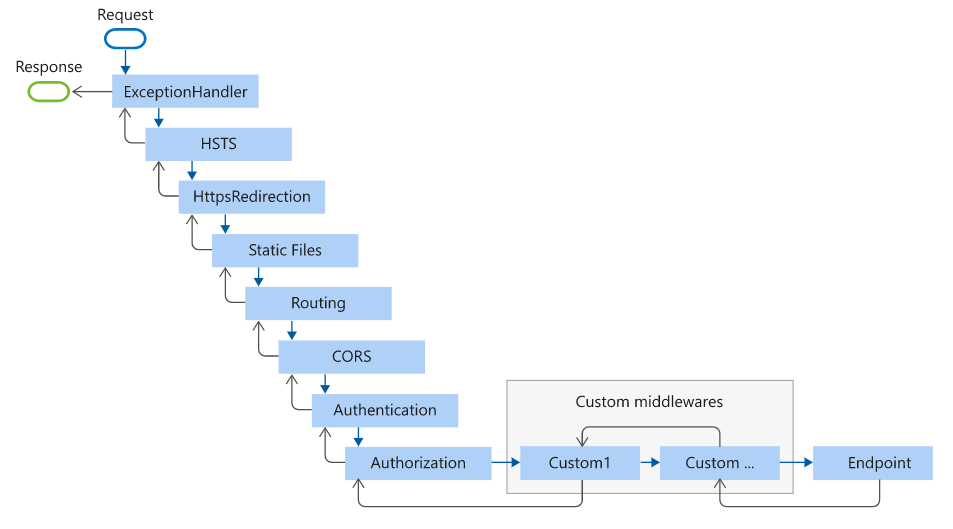
Để thêm các service này và config nó cho nó bớt cồng kềnh code và dễ maintain, chúng ta sẽ tiến hành sử dụng extension method

CORS (Cross-Origin Resource Sharing) – nó là config các nguồn mà chúng ta cho phép để truy cập tới api. Bởi vì chúng ta sử dụng nhiều domain khác nhau từ client như mobile, front-end... nên ta cần config cái này nếu không thì api không cho phép các nguồn khác truy cập vào tài nguyên





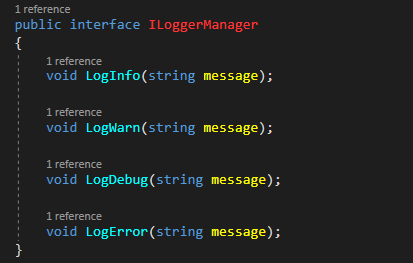
Trình tự middleware của config



1. Log service

Sử dụng thư viện của NLog để hỗ trợ việc log của chúng ta (thực ra có thể tự viết một thư viện log kiểu để ghi ra một file nào đó,... nhưng với thư viện đã có sẵn sẽ có toàn bộ tính năng ta cần mà chả cần viết lại chi cho mất thời gian).

Vì Log sẽ là một service nên ta cần nó ở mọi nơi => cần một cặp Interface-Implementation để có thể đẩy vào DI trong .NET Core

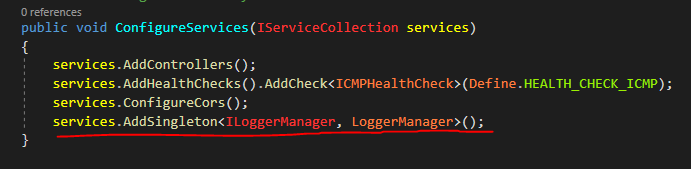


Interface sẽ là ILoggerManager, trong đấy sẽ đặt ra các rule mà implementation cần thực thi bao gồm 4 method như trên.



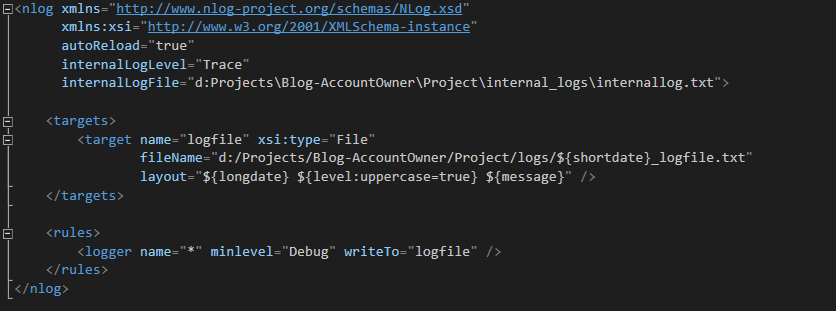
Và sau đó là implementation của nó, thực thi đúng 4 hành động mà interface quy định

Cuối cùng là ta đưa nó trở thành service và dùng trong các controller



Nếu chỉ có như vậy thì làm sao mà thằng NLog biết ghi thông tin nó log vào đâu và lưu ở chỗ nào, vậy thì việc tiếp theo ta cần làm là khi startup được khởi tạo ta đã phải config các địa chỉ để nó ghi file log rồi.

Ta tạo một file nlog.config ở root project và có teamplate như sau



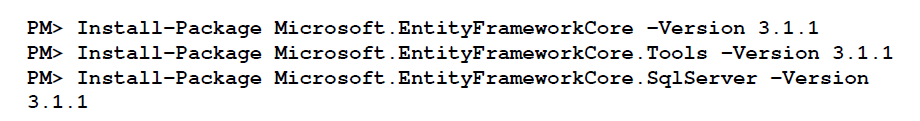
* internalLogFile: các log mà NLog viết các messages của chính bản thân nó (liệu chính bản thân thằng log có bị lỗi hay gì gì đó)
* fileName: full name của file log

1. Authentication and authorization (xác thực và cấp quyền)

# Database và migration

Lựa chọn code-first

Ta cần cài đặt sau để sử dụng EntityFrameworkCore



Với options thứ 3 ta có thể thay bằng mySql hoặc postgresql

Nhắc lại một chút về entity relationship

Với code first, chúng ta có 2 cách để config table là Annotations và Fluent API, ưu tiên là sử dụng Annotations còn thằng nào cần dấu đi hoặc chỉ sử dụng Fluent API mới config được thì mới dựng Fluent API

Relationship thì có các loại: one2one, one2many, many2one, many2many

one2one thì ít được dùng

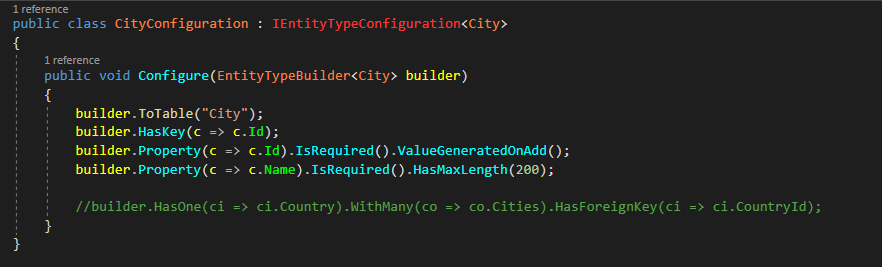
one2many và many2one là 2 thằng ngược nhau

Trong entityframework thì để config relationship người ta sử dụng 2 thứ là foregin key và navigation property (tức là property chỉ hướng tới entity mà nó có mối quan hệ)

Trong ví dụ này ta sử dụng fluent api toàn tập.

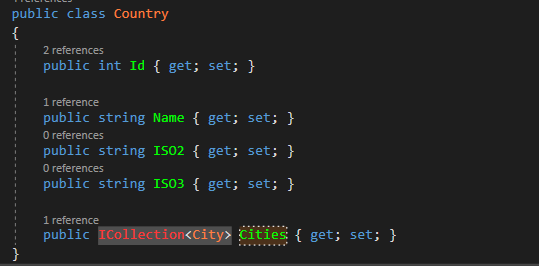
Để mà tách ra sao cho code sạch dễ maintain thì ta sẽ tạo các Config cho mỗi một Entity

Với City



Với Country





Và cuối cùng là việc migrate

Đầu tiên là cần tạo một bản migrate

dotnet ef migrations add "Init" -o "Data\Migrations"

Với “Init” là tên còn –o là option để mà di chuyển các file migrate ta tạo ra vào chỗ mà ta muốn

Sau đó để áp dụng vào database ta làm như sau

dotnet ef database update

Như vậy là database của ta sẽ update như ý muốn.

Nếu mà ta tạo migrate xong rồi mà chưa update vô database ta có thể xóa nó đi tùy ý, nhưng một khi đã áp vào database thì sẽ không thể nào mà xóa được bằng command-line

dotnet ef migrations remove

Để mà edit database thì ta chỉ cần edit trực tiếp trong fluent api hoặc annotation rồi sau đó tạo các bản migrage tiếp theo, entityframework core sẽ tự biết thêm các phần update dựa trên phần migrate Init ban đầu của ta, ghi đè các chỉnh sử ta muốn trên thằng ban đầu, và cuối cùng ta chỉ việc update là xong

# Fetching data

## Nguyên tắc và kiến trúc của một controller, API

Bối cảnh – dữ liệu của ta luốn có rất nhiều, có khi có cả hàng ngàn, hàng trăm ngàn rows. Vậy việc request từ client-side tới server mà trả về hết đống dữ liệu ấy thì hiệu năng như lz, mà cũng đéo có ram nào đủ to mà chứa được hết.

* Chúng ta cần sử dụng Page-Index và Page-Size để hiển thị trên client side, và khi đó trả về dữ liệu trên server của ta chỉ cần là số lượng tương đương với Page-Size là đủ. Vì vậy ta cần request lên server cả bao gồm Page-Size và Page-Index mà ta cần để bên server skip đi những thằng không cần thiết. 2 thằng này đóng vai trò làm thông tin chính khi gửi tới server, nhưng ngoài ra bên client-side còn muốn biết tổng số lượng, số page mà client side có thể có, rồi trạng thái của next hoặc prev... Nên ta cần một API trả về với đầy đủ thông tin trạng thái cho client-side.

Vậy thông tin trả về từ API gồm có:

* Tổng số page
* Page hiện tại
* Số lượng record mỗi page

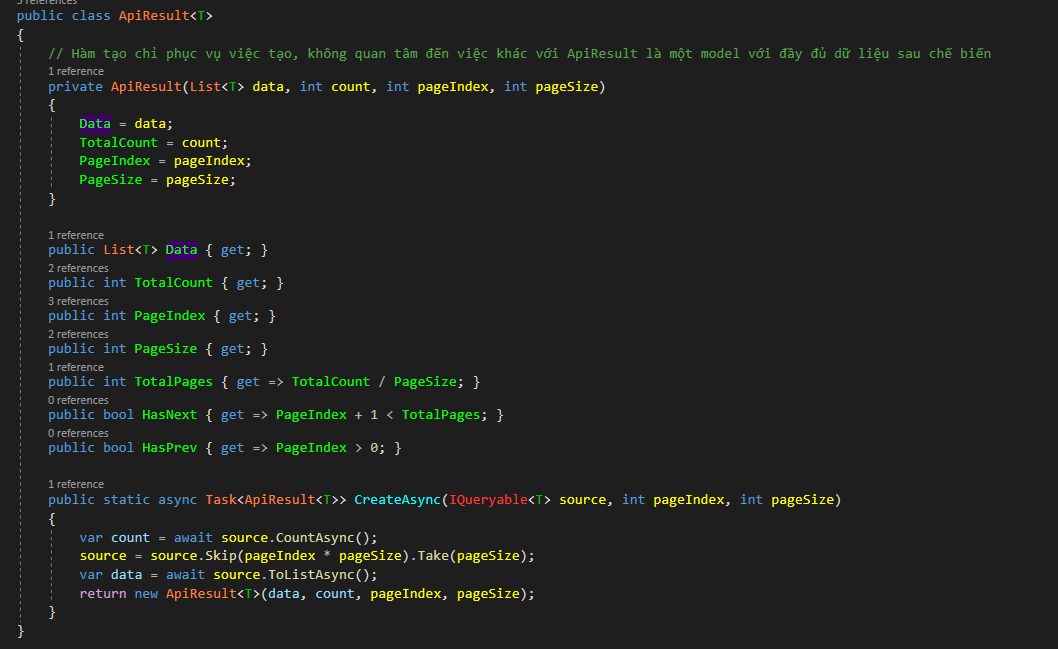
Vì có nhiều API request kiểu như thế này, nên ta sẽ tạo một module đảm nhiệm vai trò lo lắng quan tâm, tính toán đến kết quả trả về này.

### ApiResult

Đây chính là module mà ta sẽ tạo để đảm nhiệm vai trò trả về với các thông tin đó

ApiResult – có đặc điểm như sau nếu chỉ có các thông tin liên quan đến page

1. ApiResult bao luôn cả việc take care đến skip trang... nên nó sẽ cần một class xử lý logic việc này . CreateAsync
2. CreateAsync xử lý logic xong sẽ gán các giá trị rồi khởi tạo một đối tượng của ApiResult, hiểu đơn giản thì thằng này như là hàm bạn của class này, có quyền được gọi hàm tạo ở chế độ private và trả ra một object. Không có cách nào khác để có thể tạo object của class này mà không thông qua thằng này



Nếu chỉ có như thế kia thì cũng chẳng có gì thêm. Nhưng nếu mà có thêm sort với order theo column mà ta cần xử lý luôn trên server thì sao

Vấn đề đầu tiên, tên cột mà truyền từ client sang để sort cần phải đúng, nếu sai sẽ gây ra lỗi.

Đúng ở đây xoay quanh một vấn đề có nhiều khía cạnh. Có thể là chuỗi ký tự cố ý gây lỗi chương trình, phá chương trình, rồi chuỗi ký tự không đồng nhất về upper hay lower, hay không đúng với tên cột trong database. Nên trước tiên ta cần chuẩn hóa dữ liệu để đưa vào query, và thằng ApiResult đảm nhiệm luôn vai trò này. Nhưng ở đây nếu là ít logic xử lý dữ liệu thì không sao, nếu nhiều ta phải xem xét phương án chuyển thành một module riêng biệt chuyên để xử lý dữ liệu trươc skhi query



Giờ lại thêm một thứ nữa là filter theo column

làm tương tự như thằng sort

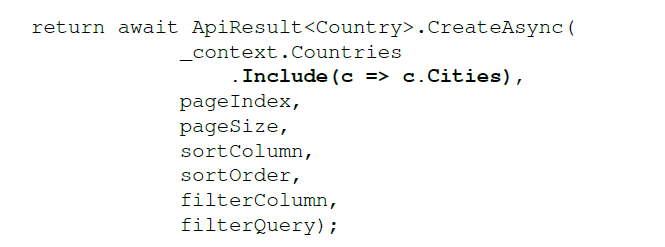
Chốt lại, với một API thì nơi mà API gọi đến nó chỉ đảm nhiệm trách nhiệm trả về, còn xử lý các logic với các thông tin thì để cho một module khác lo. Càng là phức tạp ta càng cần chia module nhỏ.

Ý tưởng về ApiResult module là khi ta có một danh sách country, một danh sách city. Chẳng nhẽ mỗi controller ta đều cần code lặp lại về logic là giống nhau, mà chỉ có khác nhau về type? Để giải quyết vấn đề này đơn giản là ta sẽ tạo một class xử lý cho cả 2 với T là generic type đóng vai trò là giả định type ở các trường hợp đó. Và ta sử dụng linq dynamic và reflection để đọc các thông tin về class mà không cần gọi một cách trực tiếp class type đó ra. Từ đó ta sẽ xử lý được cái mình muốn

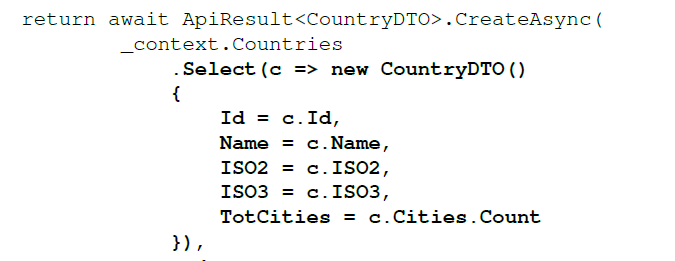
Vấn đề tiếp theo là ta luôn cần chuẩn hóa dữ liệu của mình nếu không muốn chương trình có vấn đề trong thời gian hoạt động

Một Case nữa là trường hợp, giả dụ ta muốn lấy thêm tổng số lượng city trong một country thì sao. Lúc này sử dụng DTO là hợp lý, bằng việc thêm một trường theo tính toán vào.

Vấn đề đặt ra, lấy tổng số country như thế nào cho phù hợp, chả nhé egea load rồi tìm count



Là một thằng ngu cũng nhìn ra nếu cities cực lớn thì hiệu năng ăn lz.



Xem xét thằng này thì sao. Chưa hiểu lắm khi thằng này sử dụng Select, đồng nghĩa với việc là nó sẽ xử lý phần Select này ở bên trên back-end server chứ không phải là database server. Nhưng về cơ bản nó vẫn là khả dĩ nhất, khi dùng 1 nửa query bên database 1 nửa query bên high-level vẫn ghép với nhau trong một IQuerynable

DTO = Data Transfer Object – phục vụ với mục đích duy nhất là đưa cho client-side những dữ liệu mà nó thật sự cần thiết

## Vấn đề với back-end data fetching

1. Tối ưu câu lệnh query với từng trường hợp sao cho phù hợp (nên đọc qua toàn bộ linq)
2. kiến trúc tốt các phần xử lý của API thành các module sao cho dễ xử lý
3. Vấn đề về validate, chuẩn hóa dữ liệu đầu vào từ phía client-side

Cuối cùng để áp dụng tốt thì vẫn là nguyên lý SOC.