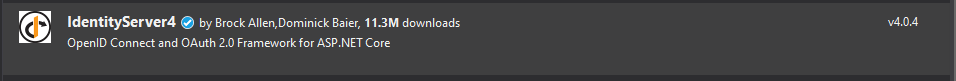
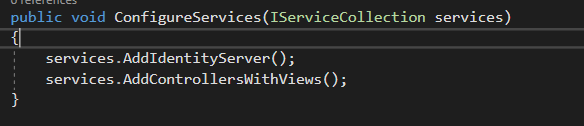
# IdentityServer

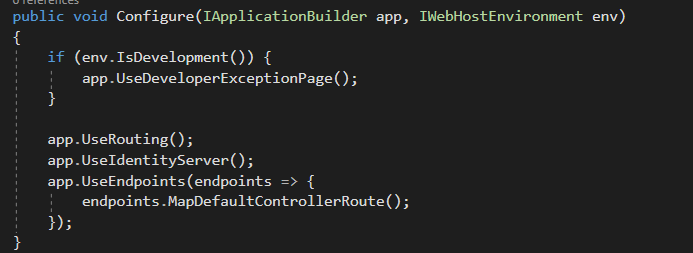
## Config

Install package sau để có được thư viện cho OIDC và OAuth



Sau khi có gói cài này rồi, ta chỉ việc thêm service của nó tới DI container và sử dụng trong middleware là có toàn bộ thiết lập của OIDC và OAuth về endpoint



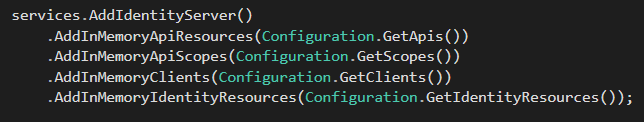


Đối với OIDC ta cần hệ thống database cho phần hạ tầng của giao thức này. Phần hạ tầng này sẽ lưu trữ lại những thứ sau

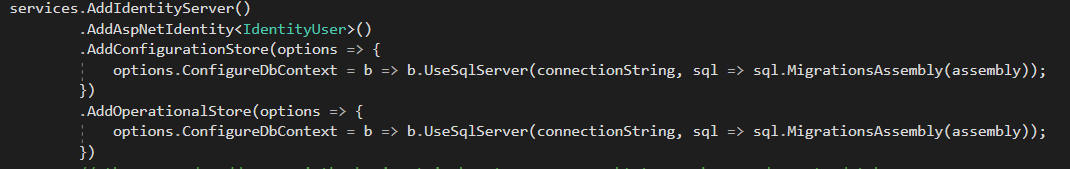
* Clients – các ứng dụng đăng kí liên kết với IdentityServer
* Apis – các Apis đăng kí với IdentityServer
* ApiScope
* IdentityResource – các scope phục vụ cho thông tin của identity

Ta có 2 phương án, một là thiết lập database với SQL Server và phương án 2 là sử dụng InmemoryDatabase (thằng này thường dùng trong quá trình dev và unit-test)

Với sử dụng Inmemory:

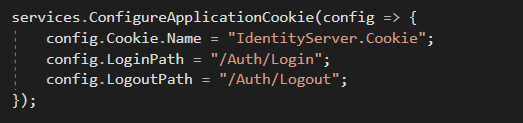


Khi sử dụng với database thực



Bản thân của IdentityServer4 package hiểu đơn giản là một bộ tools để thiết lập OIDC và OAuth. Với OIDC và OAuth ta có các endpoint, claim, scope, etc. Hiểu đơn giản thì bọn này về cơ bản là chỉ có flow giao thức về quy trình cấp access token đi như thế nào, còn việc xác thực thì chưa có. Trong ASP.NET Core ta sẽ tích hợp vơ chế authentication cookie scheme + Identity Management + MVC để đảm nhiệm vai trò cung cấp việc xác thực và quản lý user.

Bản thân thằng IdentityServer4 khi được gọi đến authorize endpoint, nó mặc định sẽ yêu cầu là xác thực dựa trên scheme mặc định là cookie authentication của .net core. Ta cần config để ghi đè lại các thiết lập mà ta muốn



Với thiết lập này, khi sử dụng các cookie authentication, nó sẽ chạy theo cái mà ta đã thiết lập.

Tiếp theo ta cần config Identity service, thằng này sử dụng EntityFramework để giao tiếp với database, và nó cũng có các built-in layer có sẵn dùng cho flow logic về quản lý user, account

Ta có mối quan hệ như sau:

IdentityServer – Authentication cookie – AspNetCoreIdentity

IdentityServer sử dụng cookie để xác thực, cookie sử dụng AspNetCoreIdentity để quản lý và trả về thông tin cho IdentityServer, nhưng bản thân thằng IdentityServer và AspNetCoreIdentity không hề nhận biết nhau, không có sự mapping về claims. Nên ta cần config để cho IdentityServer biết AspNetCoreIdentity

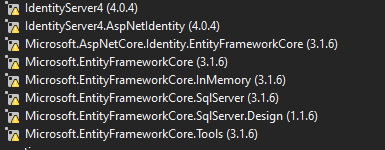
Trước tiên, tạo service Identiy cho cookie sử dụng

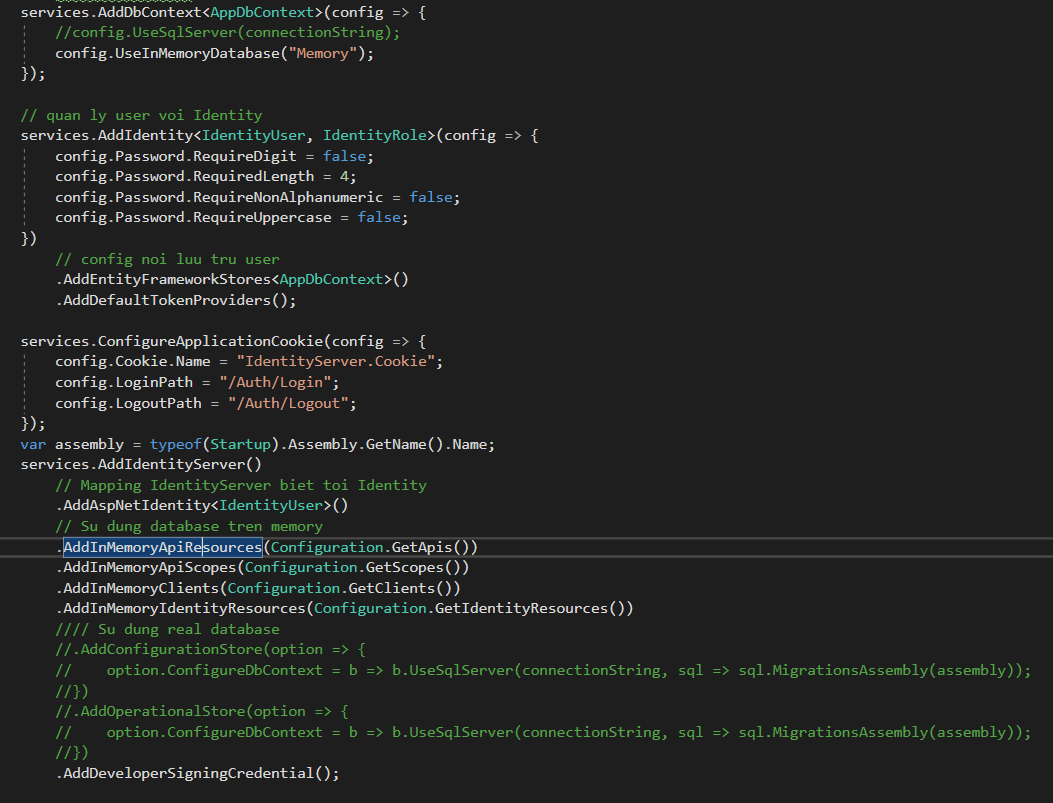
Identity hay là IdentityServer đều cần dùng đến EF Core

EFCore, EFCoreSqlServer được sử dụng để giao tiếp với database sql server

Identity.EFCore là thư viện để Identity giao tiếp với database, Identity chỉ quản lý (logic xử lý case) sau khi truy xuất dữ liệu

IdentityServer4.AspNetIdentity để mapping IdentityServer theo Identity





# ASP.NET Core Identity

Thằng này là layer phụ trách toàn bộ việc quản lý user, giao tiếp database, validate user và trả về kết quả cho Scheme

Config

services.AddIdentity<IdentityUser, IdentityRole>(config => {  
 config.Password.RequireDigit = false;  
 config.Password.RequiredLength = 4;  
 config.Password.RequireNonAlphanumeric = false;  
 config.Password.RequireUppercase = false;  
})  
 *// config noi luu tru user* .AddEntityFrameworkStores<AppDbContext>()  
 .AddDefaultTokenProviders();

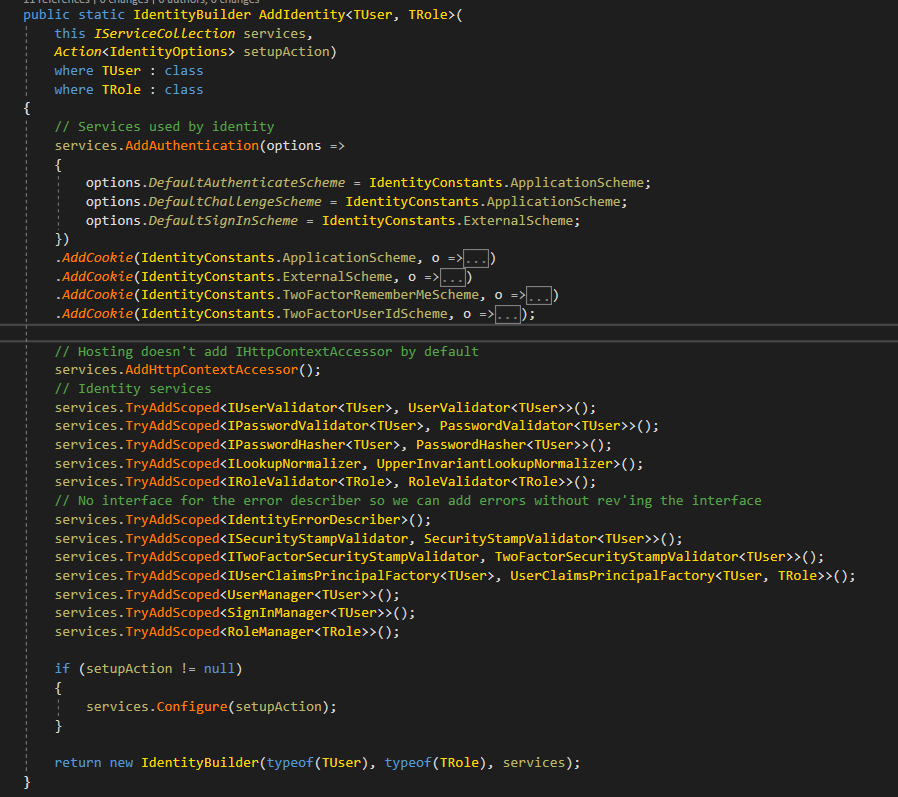
Đăng kí ASp.NET Core Identity với extension method : AddIdentity<TUser, TRole>

AddIdentity method thêm cho ta các services cần thiết cho user-manager actions như creating users, hashing passwords,... (có nghĩa nó là một layer build sẵn với các phương thức lquan đến quản lý user).

.AddEntityFrameworkStores<AppDbContext>()

Method này đăng kí implementation là EFCore cho IdentityStore

## Cùng nhìn qua một chút về Source Code của Identity



Ta có thể thấy được một điều rằng, chỉ với AddIdentity() thì trong đó nó đã thêm cả tỉ service vào DI container

IdentityOptions



Về cơ bản ta hiểu, hệ thống được tạo lên từ các services, từ các services nguyên thủy sẽ cho ra các services phức tạp hơn với các module tương ứng để sử dụng

Với AspIdentity chúng ta có vài tính năng như

* Reset Password
* Email Confirm
* Two-Step Verification (thằng này thì generate một chỗi code để đăng nhập lần 2)

Đặc điểm chung của 3 thằng này thì trong phần xử lý action, bọn chúng đều dựa vào userManager trao cho một Token và gửi tới email, sau đó ta vào trong email click vào đường link và nó dẫn ta tới action handler với parameter đi kèm là token đó. Và thế là ta có thể sử dụng token cùng với tên email để xác nhận, token lúc này coi như là mật khẩu.

# RESTful Web Services

RESTful Web Services là REST Architecture dựa trên Web Services. Trong REST Architecture mọi thứ là resource.

REST viết tắt của Representational State Transfer. REST là web xây dựng trên kiến trúc sử dụng HTTP protocol. Nó xoay quanh resource nơi mà mọi component là một resource và một resource được truy cập bởi một interface chung sử dụng giao thức HTTP.

Trong REST architecture, một REST server cung cấp khả năng truy cập tới các resources và REST client truy cập và chỉnh sử resources. Mỗi resource được định danh bởi URIs/global IDs. REST sử dụng vài thứ để đại diện cho resource như text, JSON, XML. JSON là phổ biến.

# Building a GraphQL API with ASP.NET Core 2 and Entity Framework Core

GraphQL là công nghệ tương đối mới được phát triển tại Facebook và open-sourced trong năm 2015. Trong năm 2017, nó thực sự nhảy vọt.

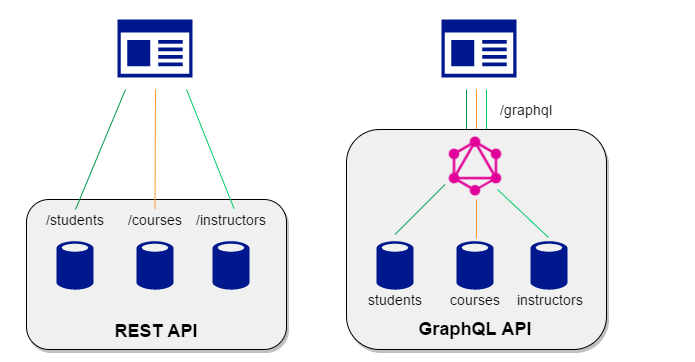
GrapQL là một ngôn ngữ query cho API, và server-side runtime cho việc queries bằng cách sử dụng một hệ thống type bạn định nghĩa cho data của bạn. GraphQL không bị bó chặt với database nào hoặc storage engine.

## Benefits of GraphQL

Trước khi đi sâu vào code, cần chỉ ra một số vấn đề cơ bản với REST mà GrapQL giải quyết và tại sao chúng ta có thể muốn chọn thay vì REST truyền thống trong một số trường hợp nhất định.

### GraphQL Is Less Chatty than REST

REST cần nhiều cuộc ghé thăm từ client tới resource endpoints trên server để lấy dữ liệu cần để render một view or page trong app của chúng ta



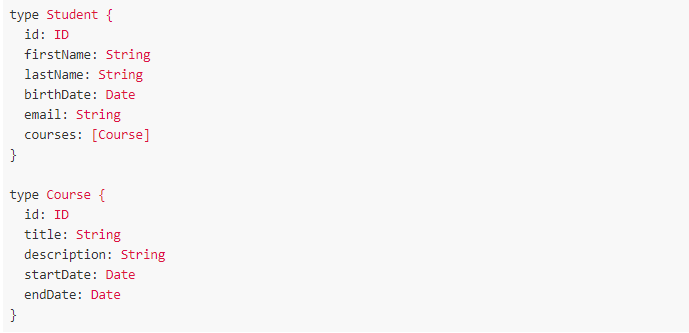
GraphQL giải quyết vấn đề này bằng việc cho phép client tạo một single query gọi tới các functions có liên quan trên server để xây dựng một response với multiple resources trong một single request.

### Endpoint Overload

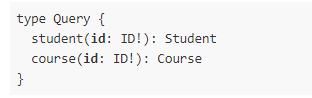
Có quá nhiều endpoint khi chương trình nó to lên

Trong GraphQL, chúng ta có thể định nghĩa Query type trong Schema của chúng ta để lấy bất cứ việc móc nối các types trong hệ thống của chúng ta chỉ thông qua một endpoint ví dụ: /api/graphql

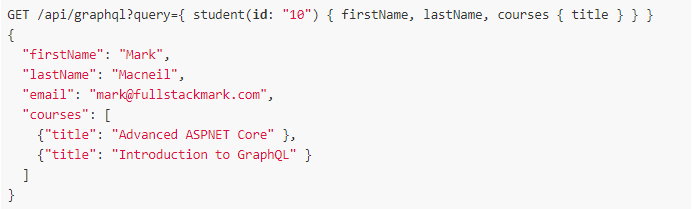
Trong schema của chúng ta có lẽ có Student và Course



Tiếp theo chúng ta cần tạo một Query type trong schema để lấy bất kì Student hoặc Course



Bây giờ chúng ta có thể request thông quan GraphQL tương tự tới RESTful. Tuy nhiên, chỉ cần gửi một endpoint là đủ



### Overfetching/Underfetching Data (lấy quá nhiều/ quá ít dữ liệu)

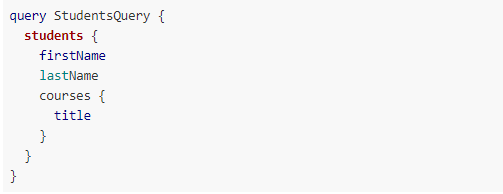
Bởi vì REST APIs trả lại cấu trúc dữ liệu cứng nó rất khó để thiết kế một API flexible đủ để đáp ứng như cầu dữ liệu chính xác của mọi clients. Điều này dẫn đến 2 điều kiện được gọi là Overfetching và Underfetching.

Overfetching cõ nghĩa là client lấy nhiều hơn thông tin trong một số trường hợp cần thiết. Xem xét kiểu Student bên trên trong một trường hợp cụ thể nơi mà chúng ta displaying một danh sách students, chúng ta chỉ cần id và name. Tuy nhiên, nếu endpoint cho /students trả về nhiều hơn là điều không cần thiết tạo ra tiêu phí tài nguyên của ta

Mặt khác, khi mà endpoint không thể cung cấp tất cả dữ liệu mà client yêu cầu thì ta phải thêm request để lấy thêm dữ liệu cần. Điều này được biết như là underfetching hoặc n+1 request. Ví dụ chúng ta cần một danh sách students và các courses của họ, nhưng /students không có thông tin về course nên ta sẽ phải request thêm lần nữa /students/:id/courses để lấy dữ liệu course (n+1)

GraphQL giải quyết vấn đề này

Trong ví dụ này, chúng ta có thể lấy data trong một shape mà chúng ta yêu cầu trên client bởi việc submitting một query cái mà nhìn dư này:



The GraphQL service sẽ trả về chính xác kết cấu chúng ta đã chỉ định không nhiều, không ít hơn

