نکات برای خودم برای پروژه دفترچه تلفن برای نوین پرداز مشهد :

بخش عمومی و کلیات :

* حتما در زمانی که **کدنویسی همه ی بخش ها تمام شد** ، ngen را برای تمام اسمبلی ها (و تمام فایل های dll) در تمام لایه ها و پروژه ها ، مخصوصا برای 3 فایل DataAccess.dll و EntityFramework.dll و EntityFramework.SqlServer.dll که در لایه و پروژه ی DataAccess قرار دارند را هم برای نسخه ی 32 و هم 64 بیتی نصب کن تا سرعت اجرای کد ، بیشتر شود .

برای نصب هم CMD را بصورت admin اجرا کن و مثلا برای نصب ngen برای نسخه ی سیستم عاملِ هدفِ 64 بیتی برای فایل DataAccess.dll ، دستور زیر را در آن بده :

C:\Windows\Microsoft.NET\Framework64\v4.0.30319\ngen.exe install "E:\Project\Visual Studio\C#.Net\Saved Project\0 Important Project\PhoneBookWPF\DataAccess\bin\Debug\DataAccess.dll"

و برای نصب ngen برای نسخه ی سیستم عاملِ هدفِ 32 بیتی برای فایل DataAccess.dll ، دستور زیر را در آن بده :

C:\Windows\Microsoft.NET\Framework\v4.0.30319\ngen.exe install "E:\Project\Visual Studio\C#.Net\Saved Project\0 Important Project\PhoneBookWPF\DataAccess\bin\Debug\DataAccess.dll"

و همینطور برای بقیه ی فایل ها هم این دستورات را بده .

برای اینکه بررسی کنی که آیا برای فایل مورد نظر در مسیری که دادی ، نصب شد یا نه ، دستور زیر را بده (دستور زیر ، برای بررسی نصب در نسخه ی 64 بیت هست) :

C:\Windows\Microsoft.NET\Framework64\v4.0.30319\ngen.exe display DataAccess

- برای ارتباط دو ماژول ، از الگوی طراحیِ "Factory" (نه الگوی FactoryMethod) یا "Service Locator" استفاده کنید .

البته الگوی Factory احتمالا مناسب تر از الگوی Service Locator باشد و پیشنهاد میشود .

منظور از Service در اینجا ، همان ماژول در برنامه نویسی هست .

- در هر پروژه ، پوشه بندی را بر اساس "ماژول ها" انجام بده .

در صورت نیاز ، درون هر پوشه ی مربوط به هر ماژول ، پوشه هایی با نام "Class" و "Interface" بساز

و کلاس ها و اینترفیس ها را درون هر پوشه ی مربوطه بریز .

=============================================================================================================

لایه ی View :

بخش تنظیمات :

- برای دفترچه تلفن ، 2 گزینه ی انتخاب استایل روشن و تاریک بذار .

=============================================================================================================

لایه ی ViewModel :

* در اینترفیس IBindingRelatedOperations ، بجای پروپرتی هایی از نوع ICommand ، احتمالا باید متد بذاری که اطلاعات مربوطه را از لایه ی مدل دریافت کنه و برگردونه .

چون کامندها ، مثل رویداد (توسط دلیگیت) ، فقط اطلاع رسانی انجام میدهند و برای برگرداندن اطلاعات ، مناسب نیستند و (هر چند میتوانند این کار را انجام دهند اما) کار را سخت میکنند .

=============================================================================================================

لایه ی Model :

- try catch را در نخ های جدیدی که در ماژول PhoneBook در این لایه نوشتی را انجام بده .

- برای ماژول "اعتبار سنجی" ، الگوی طراحیِ "Specification" را پیاده سازی کنید .

- ماژولی که در اون برنامه ی قبلیِ دفترچه تلفن نوشتی که در لایه ی Business Logic ، شماره تلفن (چه تلفن ثابت یا همراه)

را ارزیابی میکرد (که هم 8 و هم 11 رقم را قبول میکرد)، اگر همون را در اینجا استفاده میکنی ، دقت کن که در این برنامه ،

فقط شماره همراه که 11 رقم هست ، داریم (شماره تلفن خانگی که 8 رقم بود را نداریم) .

همچنین در آن برنامه ی قبلی ، ارزیابی "کد ملی" انجام شد . اما در این برنامه ، این گزینه را نداریم و بجایش "کد پستی" داریم

و دقت کنید که این دو را با هم اشتباه نگیرید . هر چند که هر دو ، 10 رقمی هستند .

=============================================================================================================

لایه ی DataAccess :

* در Repository ، متدهای Add و Edit و Remove را فقط بصورت lazy loading بنویس .

یعنی بصورت eager loading ننویس و بنابراین پارامتر idPropertyName را حذف کن .

یعنی هر موجودیت ، هر بار باید برای خودش بصورت مجزا این عملیات را انجام دهد . یعنی مثلا فرض کنید که یک رکورد از موجودیت پدر و چند رکورد از موجودیت فرزند را میخواهم اضافه (Add) کنیم . برای اضافه کردن اینها ، یکبار ، یک رکورد را به موجودیت پدر ، و بعد چند رکورد را به موجودیت فرزند اضافه کنیم.

* همچنین متدهای جستجو (مثل GetAll و Find) را بصورت جنریک بنویس (علاوه بر جنریک ای که خود کلاسِ Repository داره ، این متدها را هم جنریک کن) و جستجو را فقط بر اساس نوع جنریک در متد (نه توسط نوع جنریک کلاس Repository) انجام بده .

چون این قابلیت باید وجود داشته باشه که همزمان با حذف و اضافه و ویرایش کردن در یک نوع ، بشود که یک نوع دیگر را در همان شی از Context و Repository ، جستجو کرد .

* همچنین 2 حالت برای بارگذاری برای متدهای جستجو (مثل GetAll و Find) بنویس . یکی که بصورت عادی و استاندارد ، بصورت lazy loading و دیگری بصورت eager loading باشد .

چون هر کدام ، با روش متفاوتی انجام میدهند ، متدها را overload نکن . بلکه دو متد ، با نام های مختلف بنویس . مثلا نام یکی FindByLazyLoading و نام دیگری FindByEagerLoading باشد .

متدهایی که بصورت eager loading کار جستجو را انجام میدهند ، پارامترِ اضافه تر ای بنام idPropertyName که از نوع string هست (که مقدار پیش فرض "Id" را هم دارد) را هم داشته باشند .

* متدهای جستجویی که بصورت lazy loading کارشان را انجام میدهند ، بصورت پیش فرض ، در اینترفیس استانداردِ Repository ، یعنی در اینترفیسِ IRepository<TEntity> تعریف شوند اما آن متدهایی که بصورت eager loading کارشان را انجام میدهند ، در اینترفیس مجزا (مثلا بنام IEagerLoading) تعریف شوند و اینترفیسِ RepositoryBase<TEntity> ، از آن اینترفیس ، مشتق شود . چون ممکن است که در همه جا ، پیاده سازی و استفاده از eager loading ، نیاز نباشد .
* در متد Edit ، این را تست کن که کد و مقدار پروپرتیِ context.Entry(entityInDb).CurrentValues را درون متغییر بریز و ببین آیا همه ی navigation property ها را در آن موجودیت (مثل PersonEntity) ، میتونه مقادیرشون را نشون بده یا نه . برای اینکه بررسی کنی که اگر نشون نداد ، مثلا ببینی روشی هست که درون این اشیاء و پروپرتیِ CurrentValues ، مثل مثلا متد Include ، بصورت Eager Loading ، آن موجودیت را بارگذاری کنی یا نه .

همچنین برای خروجی متدِ context.Entry(entityInDb).CurrentValues.SetValues درون این متدِ Edit هم این را بررسی کن .

* به هر حال نباید شی ای که از کلاس PhoneBookDbContext میسازی ، اگر در یک متغییر ذخیره میکنی ، نباید نوع اون متغییر ، از خود همین کلاسِ PhoneBookDbContext باشه . به هر حال باید توسط اعضای یک اینترفیس ، به این کلاس دسترسی پیدا کنی (البته مشکلی نداره که این شی را بدون ریختن در متغییری بهش دسترسی پیدا کنی) .

احتمالا این طوری عمل کن که در اینترفیس مربوط به کلاس Repository ، متدهای مربوط به کلاسِ PhoneBookDbContext را که میخوای فراخونی کنی (مثل متد InitializeDatabaseAsync و Dispose) را درون این اینترفیس قرار بده و توسط متدهای این اینترفیس ، به متدهای کلاس PhoneBookDbContext دسترسی پیدا کن .

- یک کلاسی بساز که از کلاس CreateDatabaseIfNotExists<PhoneBookContext> ارث بری کنه و متد Seed را override کنه و در این متد ،

داده های جداول و موجودیت های ProvinceEntity و CityEntity را پر کن چون این متد ، فقط برای اولین باری که دیتابیس

موجود نباشد ، اجرا میشود .

همچنین در متد سازنده ی کلاس PhoneBookDbContext ، متد استاتیکِ Database.SetInitializer را که فقط برای پیکربندی هست ،

فراخوانی و شی ای از این کلاسی که ساختید ، بسازید و به ورودی این متد بدهید .

همچنین یک متد Async در کلاس PhoneBookDbContext بسازید که متد شی گرای this.Database.Initialize را در Task جدیدی فراخوانی کند

تا متد Initialize که دیتابیس جدیدی را میسازد ، درون نخ دیگری بسازد و این متدی که ساختید هم شیِ Task را برگرداند (مثلا نام متد

را CreateDatabaseAsync بگذارید) .

در بیرون از این کلاس هم وقتی برنامه اجرا شد ، شی ای از کلاس PhoneBookDbContext بسازید و متد CreateDatabaseAsync را فراخوانی کنید .

در لایه های دیگه ، مخصوصا در لایه ی Model (و همچنین View) ، اگر درخواستی به EF ارسال شود که قبل از آن ، دیتابیس ساخته نشده باشد ،

احتمالا خطای System.Data.SqlClient.SqlException پرتاب میشود که حتما مدیریت و به کاربر پیغامش را بده .

- مقدار پیکربندی متد استاتیکِ SetInitializer ، همون روی مقدار پیش فرض اش که CreateDatabaseIfNotExists هست ، بماند .

چون نسخه ی اول هست و model ئه موجودیت های EF مان تازه ساخته میشود و تغییری نمیکند ، لازم به استفاده از migration هم نیست .

- رابطه ی موجودیت PersonEntity را با موجودیت AddressEntity ، رابطه ی چند به چند بگیر (برای هر فرد ، ممکنه چند آدرس ذخیره کنید) .

- همه ی جدول ها به index نیاز ندارند . پس ، مشتق بودنِ اینترفیسِ IAttributeConfigurationBase را از اینترفیسِ -

IAttributeIndexConfiguration ، خارج کن و مستقلا برای هر کلاس موجودیتی که نیاز بود ، اینترفیسِ IAttributeIndexConfiguration -

را پیاده سازی کن .

- برای اینترفیس هایی که خالی هستند و خودشون ، عضو ندارند مثل اینترفیس های IEntityConfiguration و IAttributeConfiguration -

به آخر این اینترفیس ها ، کلمه ی "Base" اضافه کنید .

همچنین ، اینترفیس ای بنام IAttributeUniqueConfiguration بسازید که متدی بنام SetAttributeUnique داشته باشد .

این اینترفیس ، نباید فرزند اینترفیسِ IAttributeConfiguration شود چون فقط بعضی از کلاس ها ، این متد را دارند و

این اینترفیس را پیاده سازی میکنند .

همچنین در متد مورد نظر در کلاس Facade هم ، شیِ اینترفیسِ IEntityConfiguration را به اینترفیسِ IAttributeUniqueConfiguration تیدیل

کن و اگر null نبود ، متد مورد نظر در این اینترفیس را هم اجرا کن .

--------------

- موجودیت هایی که باید کلید خارجی (از نوع int) در آنها تعریف شود :

MobileNumberEntity : که به کلید اصلی در موجودیت PersonEntity اشاره کند .

CityEntity : که به کلید اصلی در موجودیت ProvinceEntity اشاره کند .

- برای index گذاشتن روی کلید خارجی ، پس ، در کلاس های مربوط به موجودیت های جداولی که نیاز به کلید خارجی دارند ،

پروپرتی هایی (احتمالا همه شون از نوع int) تعریف کن و اینها را به کلید خارجی تبدیل و روی اینها index بذار .

چون اگر از طریق دستورات sql بخوای روی کلید خارجی ها ، index بذاری ، باید اینها را در متدی در کلاس مشتق DbContext تعریف کنی و

چون در این حالت ، وظایف پیکربندی ، در کلاس های مشتق EntityTypeConfiguration<TEntity> محول شده ، پس نقض Solid صورت میگیرد .

- در همه ی موجودیت هایی (یا اغلب موجودیت هایی) که کلید خارجی در آنها تعریف شده ،

آن ستونِ کلید خارجی را به عنوان ستون index تنظیم کن .

- برای فراخوانی متدهای کلاس EntityConfiguration از متد DbContext.OnModelCreation ، از کلاس میانی ای که الگوی طراحی Facade را دارد ،

استفاده کنید . یعنی درون متدِ DbContext.OnModelCreation ، متد مورد نظر در کلاس میانی Facade ، و از آن متد ، تمام متدهای کلاس های

EntityConfiguration را فراخوانی کنید .

- اینترفیس IEntityConfiguration ، از چندین اینترفیس ارث بری کنه که هر کدوم از اونها ،

متدهای SetTableName و SetAttributeDataType و SetAttributeRequire را داشته باشند برای اجرای اصل "جداسازی اینترفیس ها" در solid