1. چرا از LINQ استفاده می کنیم؟
   1. یک کد برای هر دیتابیس. مهم نیست چه دیتابیسی داریم استفاده می کنیم، کدی که با LINQ می نویسیم روی همه دیتابیس ها جواب میده.
   2. strongly typed. یعنی مثل raw query نیست که خود کوئری مورد نظر که برای دیتابیسمون میخوایم بنویسیم رو ببریم داخل یه string بلکه با کد می نویسیمیش در نتیجه trace ، تست و دیباگ کردنش راحتتره.
   3. 2 نوع syntax داره؛ query که مشابه sql هست و مدل دیگه که بهش linq method یا fluent میگن. این دومی از extension method ها استفاده میکنه. توی اجرا این دوتا تفاوتی ندارن و یه خروجی رو میدن.
   4. خوندن کوئری در حالت LINQ ساده تر و قابل فهم تره؛ توی LINQ منظور بهتر منتقل میشه و راحتتر میشه فهمید که هر کوئری هدفش چیه.
2. طبق قسمت بالا ما دوتا مدل برای نوشتن LINQ داریم: Query و Fluent. از این به بعد با همین اسم به هر کدوم مدل ها اطلاق می کنیم.
3. برای قسمت Where داخل query، میتونیم علاوه بر نوشتن یه expression که حاصل یک یا چندتا شرط هست، یه متد که خروجیش bool هست رو استفاده کنیم (ورودی متد میتونه یه آبجکت از نوع کلاس داخل لیست ما باشه). به طور مشابه، fluent هم این قابلیت رو داره. حتی چیزی هست به اسم method group که کافیه ما فقط اسم متد مورد نظر رو داخل Where() بنویسیم؛ این حالت نیازی به lambda function نداره چون خود csharp میفهمه که باید یه انتیتی از نوع مورد نظر رو به عنوان ورودی به متد بده.
4. در هنگام OrderBy، اگه از type های از قبل تعریف شده مثل عدد ها، string، DateTime و... استفاده کنیم، به طور پیش فرض از IComparable تعریف شده خود csharp استفاده می کنه اما برای انواع دیگه باید یه کلاس که IComparable رو implement می کنه رو برای اون انتیتی تعریف کنیم.  
   همچنین در Query، با یه ویرگول راحت میشه بر اساس چندتا فیلد OrderBy رو انجام داد اما برای fluent اینطور نیست. اول باید OrderBy رو برای فیلد اول مورد نظر مرتب سازی بنویسیم و بعدش از ThenBy استفاده کنیم و همینطور تا جایی که میخوایم فیلد مرتب سازی داشته باشیم، اضافه کنیم.
5. deffered execution چیست؟
   1. این یه مفهوم به نظر ساده اما در عین حال پیچیده اس. کلیت اش اینه که یه کوئری، تا زمانی که واقعا نیاز نباشه اجرا نمیشه. بحث پیچیدگیش اما تو اینه که در شرایط مختلف این نحوه اجرای کوئری چطوریه. مثلا یه مدل اجرا اینه که وقتی داریم روی کوئری، loop میزنیم، بیاد و در حین loop زدن کوئری رو اجرا کنه؛ فرض کنیم یه دیتابیس از فیلم ها داریم و میخوایم فیلم هایی رو بیاریم که تام کروز توشون بازی کرده. کوئری رو می نویسیم و میذاریم تو یه متغیر. حالا اگه روی این متغیر که از نوع IEnumerable هست بیایم یه foreach بزنیم، میبینیم که کوئری داره روی دونه دونه فیلم هایی که داخل دیتابیس هستن اجرا میشه و وقتی به یه فیلم از تام کروز برسه، میره داخل بدنه foreach و بدنه رو اجرا می کنه. بعدش دوباره ادامه کوئری اجرا میشه و میره تا به فیلم بعدی تام کروز برسه و دوباره تا رسید، بدنه foreach رو اجرا کنه و همینطور الی آخر.
   2. همین دیتابیس رو اگه بعد از نوشتن کوئری، یه ToList() روش بزنیم و اینو داخل متغیر ذخیره کنیم، بلافاصله کوئری به طور کامل اجرا میشه و دیتا میاد داخل مموری. این ToList چه بعد از خود کوئری نوشته بشه و چه بعد از اینکه کوئری رو داخل یه متغیر ریختیم و حالا روی اون متغیر ToList بزنیم، فرقی نداره و میاد کوئری رو کامل اجرا میکنه و میریزه داخل یه متغیر.
   3. این داستان ToList برای First هم صادقه اما با یه تفاوت اونم اینه که به محض رسیدن به اولین رکورد که شرط مورد نظر رو ارضا کنه اجرای کوئری متوقف میشه.
   4. یکی از موارد کاربرد این ویژگی deferred execution اینه که میتونیم چندتا کوئری رو ترکیب کنیم. مثلا یه کوئری داخل متغیر a هست و یه کوئری دیگه داریم که داره روی کوئری a کوئری میزنه. اینجا اینطوری میشه که اگه بخوایم کوئری دوم رو روش مثلا foreach بزنیم، تا زمان رسیدن به foreach هیچ کدوم از بخش های کوئری اول یا دوم اجرا نمیشن و تازه سر foreach این کوئری ها شروع به اجرا شدن می کنن.
6. ما میتونیم شروطی که داخل where می نویسیم رو داخل متد هایی مثل First یا Last هم بنویسیم و اینطوری کوئری کوتاه تری داریم اما بسته به شرایط میتونه استفاده از این ویژگی تغییراتی ایجاد کنه؛ از بحث پرفورمنس تا نحوه اجرا و بیزنس پشت کوئری. پس اینطوری نیست که همیشه صرفا برای راحتی از این مدل استفاده کنیم.
7. متد Chunk چه کار می کند؟
   1. این متد لیست مورد نظر را به بخش های با تعداد درخواست شده (پارامتر int ورودی) تقسیم می کند. در واقع خروجی آن یک IEnumerable<Entity[]> است که هر دیتا در این IEnumerable یک آرایه به طول پارامتر ورودی می باشد. اگر تعداد کل دیتا بر این پارامتر بخش پذیر نباشد، همه بخش ها به غیر از بخش آخر دارای طول مورد نظر بوده و فقط بخش آخر به تعداد باقی مانده دیتا دارد.
   2. اگر بعد از Chunk، از Select استفاده کنیم، میتونیم علاوه بر خود Chunk ها، یه index هم کنارشون داشته باشیم. مثلا این:  
      list.Chunk(4).Select((chunk, index) => new {chunk, Number = index+1}))
8. متد های Skip و Takeچی هستن و تفاوتشون چیه؟
   1. این دوتا مثل Include و Exclude هستن و هر کدوم 3 مدل دارن: عادی شون که پسوند دیگه ای نداره، Last شون که از آخر کار مورد نظر رو لحاظ می کنه و While شون که هر دیتایی که تو شرط مورد نظر صدق کنه رو روش کار مورد نظرشون رو انجام میدن.  
      حالا خود کار اینا چیه؟ متد های Take میان Include رو انجام میدن؛ یعنی میگن چه دیتایی برداشته بشه. متدهای Skip اما کار exclude رو انجام میدن و برعکسن؛ یعنی میگن چه دیتایی برداشته نشه.
   2. این متد ها به صورت ترکیبی هم میتونن استفاده بشن. مثلا اگه بخوایم 4 دیتای اول کوئری رو نداشته باشیم و بعد از اون، 7 تا دیتای بعدی (دیتای 5 الی 11 از حاصل کوئری اصلی) رو برداریم، میتونیم اول یه Skip(4) و در ادامه اش یه Take(7) داشته باشیم.
9. متد های All و Any چی هستن و تفاوتشون چیه؟
   1. هر دو یه شرطی رو درباره تمام عناصر موجود در یه لیست بررسی می کنن منتهی Any به دنبال اینه که حداقل یه عنصر پیدا بشه که این شرط دربارش صدق کنه؛ اولین عنصری که ببینه این شرط دربارش صدق می کنه، بلافاصله از حلقه بررسی میپره بیرون و true میده و در غیر این صورت false. از اونور All میخواد ببینه که آیا تمام عناصر داخل لیست، این شرطه دربارشون صدق می کنه یا نه؛ اولین عنصری که این شرط دربارش صدق نکنه، بلافاصله از حلقه بررسی میپره بیرون و false میده و در غیر اینصورت true میده.
10. متد Concat؟
    1. یه جورایی از اسمش پیداس؛ قراره عناصر داخل یه IEnumerable ای رو به یه IEnumerable دیگه اضافه کنه. مشخصا عناصر داخل هر دو IEnumerable باید از یه type باشن.
11. متد های Union, Intersect, Except برای چه کاری هستن و تفاوتشون در چیه؟
    1. فرض کنیم 2 تا لیست یا IEnumerable داریم به اسم A و B. اینا رو میشه مثل مجموعه های داخل ریاضیات در نظر گرفت. در نتیجه میتونیم مفاهیم اشتراک و اجتماع و تفاضل و... رو براشون داشته باشیم. هر کدوم از این متد ها یکی از این مفاهیم رو پوشش میدن که در ادامه هر کدوم رو توضیح میدیم:
       1. Union: این مثل اجتماع دو مجموعه اس؛ یعنی هرچی عنصر داخل مجموعه A و B هست رو توی یه مجموعه جدا بذاره و به عنوان خروجی بهمون بده. نکته ای که داره اینه که مثل قواعد مجموعه ها در ریاضیات که میگه از یه عنصر فقط یه نمونه داخل مجموعه میتونیم داشته باشیم، این متد هم به صورت پیش فرض اگه ببینه یه عنصری داخل هر دو مجموعه وجود داره، فقط یه نمونه از اون رو داخل مجموعه خروجی میذاره.
       2. Intersect: این مثل اشتراک دو مجموعه اس؛ یعنی فقط عناصری که داخل هر دو مجموعه وجود دارن رو به عنوان خروجی توی یه مجموعه جدا میده.
       3. Except: این مثل تفاضل دو مجموعه اس. اگه روی مجموعه A متد Except رو با ورودی B بنویسیم، میاد و هر عنصری که مشترکا داخل هر دو مجموعه A و B هست رو از A حذف میکنه و مجموعه حاصل رو به عنوان خروجی میده.
    2. هر سه این متد ها یه نسخه By هم دارن (UnionBy, IntersectBy, ExceptBy). تفاوت این نسخه دوم در اینه که کار هر متد رو بر اساس یه پراپرتی یا key خاصی انجام بده. یه مثال میزنیم: فرض کنیم مجموعه A از type مثلا مشتری (customer) باشه که یه سری پراپرتی مثل اسم و شماره تلفن داره. از طرفی ما یه لیست دیگه داریم (مثلا C) که شامل اسم مشتری هایی هست که میخوایم اطلاعاتشونو در بیاریم. اینجا میایم از IntersectBy استفاده می کنیم و لیست اسم ها (که string هستن) رو به عنوان ورودی میدیم و در ادامه یه func هم میدیم که پراپرتی Name از کلاس customer رو داره بر میگردونه. اتفاقی که میفته اینه که از مجموعه A، فقط مشتری هایی که داخل لیست C هستن رو به ما میده.   
       مزیتی که این نسخه By داره اینه که لازم نیست دو مجموعه مورد نظر، از یه type باشن.
    3. تمام این متد ها از IEqualityComparer هم پشتیبانی می کنن.
12. متد SelectMany؟
    1. ما میتونیم type ای داشته باشیم که داخل خودش، پراپرتی (ها)ای داشته باشه که خودش یه لیست از یه type دیگه باشه. مثلا اگه یه لیست از فیلمها داشته باشیم، هر فیلم خودش یه لیستی از بازیگرها میتونه داشته باشه. حالا اگه ما لیستی از فیلمها داشته باشیم و بخوایم به بازیگرهای هر فیلم هم دسترسی داشته باشیم (روی اون لیست های بازیگرها کوئری بزنیم)، از متد SelectMany استفاده می کنیم.
    2. این متد ورودی اولش مشخص میکنه که کدوم پراپرتی درونی هست که لیسته و میخوایم بریم سراغ اون. ورودی بعدی، یه func هست که دوتا ورودی داره: اولی میشه شئ ای از type لیست اصلی (فیلم) و دومی میشه شئ ای از type لیست درونی (بازیگر). بعدش داخل بدنه func مینویسیم که چه خروجی ای میخوایم بهمون داده بشه.
13. انواع join ها در LINQ و کاربرد هر کدام؟
    1. اصل join که همون برگردوندن عناصر بین دو لیست بر اساس رابطه بین شون هست. حالا دو مدل join داریم:
    2. Join: این عین Inner join توی sql هست؛ یعنی اشتراک بین دو لیست رو برمیگردونه. اینم یه extension method ئه و روی یه لیست اعمال میشه که برای نوشتنش، اول باید روی لیست اول اعمالش کنیم بعدش باید 4 تا ورودی بدیم:
       1. اسم لیست دوم که قراره با این لیست join بخوره.
       2. یه پراپرتی از لیست اول که رابط با لیست دومه
       3. یه پراپرتی از لیست دوم که رابط با لیست اوله.
       4. یه func که ورودیش میشه شئ از لیست اول و شئ از لیست دوم و خودش در نهایت باید خروجی مد نظر ما رو داشته باشه.

این وسط اگه لازم باشه که join بر پایه بیش از 2 پراپرتی انجام بشه، کافیه که ورودی های دوم و سوم رو از نوع یه کلاس یا یه anonymous type بدیم که ترتیب تعریف کردن پراپرتی های داخل اونا، مشخص میکنه که کدوم پراپرتی از لیست اول باید با کدوم پراپرتی از لیست دوم، نظیر بشه.

* 1. GroupJoin: این میاد که معادل Left join توی sql باشه؛ یعنی از لیست اول همه رو بیاره و اگه از لیست دوم چیزی بود که بشه با چیزی از لیست اول همخونی داشته باشه، اینا رو به هم میچسبونه و اگه نبود، فقط همون عنصر به تنهایی بیاد.  
     اینم مثل join 4 تا ورودی می گیره که معادل همونا رو اینجا هم داریم. چیزی که فرق میکنه، بحث نحوه خالی یا پر کردن عناصر از لیست دوم در صورتی که به عنصری از لیست اول بتونن متصل بشن یا نشن، هست. برای این موضوع، کافیه که یه SelectMany استفاده کنیم که ورودی اولش، میشه همون عنصر دوم (در واقع پراپرتی از نوع عنصر دوم) اما در ادامه اون، یه .DefaultIfEmpty() هم میاد. اینطوری اون پر یا خالی بودن عنصر دوم تعیین میشه.

1. تفاوت IEnumerable و IQueryable؟
   1. تفاوت این دوتا در اصل به نحوه اجرا و کاری هست که انجام میدن. به طور کلی ما کوئری رو معمولا روی یه دیتابیس ای انجام میدیم. برای اتصال به اون دیتابیس و تبدیل کد C# به زبان اون دیتابیس، نیاز به یه فریمورک مثل Entity framework داریم. کوئری ای که ما داخل EF می نویسیم، برای اینکه بتونه چندتا extension method رو بگیره تا لایه های مختلف کوئری رو بشه روش نوشت، خروجی های این extension method ها (که نمونه های زیادی از اونا رو بالاتر معرفی کردیم) از نوع IQueryable میشن؛ بر خلاف حالت عادی که روی یه لیست اگه اجرا بشن از نوع IEnumerable هست.
   2. در واقع، ذات کلی این دوتا یکی هست منتهی روی جاهای مختلف عمل می کنن. هر دو اجازه میدن که لایه های مختلف کوئری رو ایجاد و اضافه کنیم و تا زمانی که در انتهای کوئری، از متد هایی مثل First یا ToList استفاده نشده، این کوئری اجرا نمیشه. اما وقتی که روی دیتابیس قراره کوئری زده بشه، از IQueryable استفاده می کنیم (که در واقع اون کوئری رو به زبان دیتابیس مورد نظر، ترجمه می کنه) و هر وقت قراره in-memory و روی یه لیستی که از قبل داخل رم هست کوئری زده بشه، از IEnumerable استفاده می کنیم.
   3. یه مثال ریز: فرض کنیم یه سری دیتا رو میخوایم از دیتابیس بگیریم. با استفاده ازEntity framework، اتصالات کد C# مون رو به دیتابیس فراهم می کنیم. حالا با استفاده از DbSet و extension method های LINQ، کوئری رو می نویسیم و توی یه متغیر میذاریم. چون این کوئری، سمت دیتابیس هست، نوع این متغیر میشه IQueryable منتهی هنوز این کوئری اجرا نشده. چون یه سری دیتا قراره بگیریم، بعد از تکمیل کوئری، یه ToList اضافه می کنیم که این باعث میشه کوئری ما سمت دیتابیس اجرا بشه.   
      اون ToList، یه لیست از انتیتی مورد نظر ما رو بر می گردونه منتهی میدونیم که کلاس List، خودش داره IEnumerable رو implement می کنه پس در اصل یه IEnumerable هم هست. پس الان ما حاصل کوئری سمت دیتابیس رو داخل رم سیستم داریم.  
      حالا اگه لازم باشه روی این حاصل کوئری هم یه سری کوئری جداگونه بزنیم، ایرادی نداره و میتونه انجام بشه منتهی این یکی دیگه از نوع IQueryable نیست و IEnumerable ئه.
2. ترتیب و نحوه اجرا در یک کوئری به چه صورت است؟
   1. اول اینطور به نظر میاد که اگه ما داریم یه کوئری انجام میدیم که لایه (بخونیم extension method ها)ی مختلف که فیلتر های متفاوتی رو روی کوئری دارن اعمال می کنن، اولین لایه اجرا میشه و بعدش جواب اون رو لایه بعدی میخونه و روش کار خودشو انجام میده و الی آخر.  
      اما در واقعیت اینطور نیست. در اصل، اولین متدی که اجرا میشه آخرینی هست که روی کوئری اعمال کردیم (معمولا این متد های آخر مواردی مثل Single, First, ToList و... هستن). هر کدوم این متد ها چون روی یه IEnumerable اجرا میشن، یه حلقه داخل خودشون دارن. برای متد هایی مثل First، بهینه ترین حالت زمانی هست که به محض رسیدن به اولین جواب درست، ادامه اجرای کوئری متوقف و نتیجه برگردونده بشه. حالا اینجا متد هایی مثل Where از کلمه کلیدی yield استفاده می کنن که هر وقت داخل خودشون به یه عنصر درست برخورد کردن، ادامه اجرای خودشون رو متوقف میکنن و این عنصر رو به لایه بعدی میفرستن. اگه لایه بعدی باز هم یه فیلتر باشه، باید از اونم گذر کنه اما اگه لایه بعدی، همون لایه آخری باشه و این عنصر، همون عنصر باشه، اجرای کوئری به اتمام میرسه و این خیلی بهتر از اینه که هر لایه، یه دور کامل کار خودش رو انجام بده و نتیجه رو به لایه بعدی بده تا اونم کامل کار خودشو انجام بده و الی آخر.  
      خوبی yield اینه که تو بدترین سناریو با حالت عادی اجرا که هر لایه کار خودشو کامل انجام بده، برابری می کنه در نتیجه اینطور نیست که در سناریویی بتونه پرفورمنس رو کم کنه.
   2. برای IQueryable میتونیم از ExpressionFormatter.Visit استفاده کنیم که نشون میده مراحل و ترتیب اجرا توی یه کوئری به چه صورته.
3. PLINQ چیه؟
   1. این همون LINQ هست منتهی به صورت parallel (ParallelLINQ). در واقع همون کار هایی که با LINQ انجام میدیم رو PLINQ هم انجام میده اما با این تفاوت که این یکی به صورت parallel اجرا میکنه. این یعنی سرعت بیشتر و استفاده از منابع سخت افزاری.
   2. عمل parallelization با متد AsParallel در ابتدای کوئری انجام میشه و به تعداد core های داخل cpu، بخش ایجاد میکنه که هر بخش روی یه core اجرا میشه و در نهایت توی یه خروجی مشترک به دست ما میرسه. منتهی این خروجی بر خلاف حالت غیر parallel، مرتب شده نیست و به هم ریخته اس. چاره اش اینه که یه AsOrdered هم اضافه کنیم. (اگه نیاز به مرتب سازی داریم)
   3. اگه بخوایم تعداد core ها (یا در واقع thread ها چون هر بخش که به یه core اختصاص پیدا میکنه در اصل داخل یه thread ریخته میشه) رو محدود کنیم، با متد WithDegreeOfParallelism میتونیم بگیم که چندتا thread ساخته بشه.
   4. ممکنه لازم باشه کاری که بعدا با دیتای خروجی از کوئری مون میخوایم انجام بدیم رو هم parallel اجرا بشه؛ انگار که کل کار (از واکشی دیتا تا انجام کارهای مورد نیاز بعدی) مستقیما به صورت parallel اجرا بشه. برای اینکار کافیه که از ForAll استفاده کنیم و بعدش یه Func به عنوان ورودی بهش بدیم که این میشه همون کاری که میخوایم بعد از واکشی دیتا انجام بشه. در نتیجه این کار هم به صورت parallel اجرا میشه.
   5. زمانی که داریم PLINQ استفاده می کنیم، خروجی هر thread میاد و داخل thread اصلی
   6. برنامه قرار میگیره. ما میتونیم با متد WithMergeOptions نحوه قرار گیری داخل thread اصلی رو هم مدیریت کنیم. مثلا با مقدار NotBuffered، زمانی که یه thread کارش تموم میشه، همون موقع اونو به thread اصلی میده (تو این حالت، بعد پاس دادن خروجی به thread اصلی، ادامه کار روی کوئری هم سمت thread اصلی انجام میشه و منتظر اتمام کل کوئری نمی مونه)؛ در مقابل FullyBuffered صبر میکنه تا کل thread ها کارشون تموم بشه و بعد همه رو داخل thread اصلی میاره. AutoBuffered هم اختیار رو به LINQ میده که خودش تصمیم بگیره چه روشی مناسبتره.
   7. اگه وسط کوئری به موقعیتی خوردیم که به هر دلیلی فکر می کنیم باید به صورت عادی (sequential) اجرا بشه، کافیه قبل از اون قسمت AsSequential بذاریم. در واقع کنترل می کنیم که کدوم قسمت از کوئری Parallel و کدوم قسمت Sequential باشه.
   8. یه نکته ریز درباره PLINQ اینه که اگه ما داریم با LINQ یه کوئری سمت دیتابیس می نویسیم، این کوئری دیگه قابلیت های PLINQ روش جواب نمیده؛ یعنی روی کوئریِ مستقیم در سمت دیتابیس تاثیری نمیذاره و فقط زمانی که دیتا به صورت In-memory هست میتونه parallel رو روی کوئری اعمال کنه. همچنین به خاطر اینکه DbSet در EF، thread-safe نیست ممکنه مشکلاتی ایجاد بشه.
4. موقع استفاده از PLINQ چه مواردی رو باید در نظر گرفت؟
   1. تعداد هسته cpu
   2. MergeOption ها (NotBuffered, FullyBuffered,...)
   3. آیا واقعا کاری داریم انجام میدیم که محاسبات داره یا طولانی هست (که ارزش داشته باشه تیکه تیکه اجراش کنیم)؟
   4. چه میزان از این کار رو میشه parallel کرد؟ (کل کار، فقط واکشی دیتا، فقط کار های بعد از واکشی دیتا و...)
   5. مهمه که متد ها و کارهایی که انجام میخوایم بدیم، thread-safe باشن و تا حد امکان استفاده ازشون محدود باشه (چون sync کردن این متد ها و خود thread-safety باعث کندی میتونن بشن)
   6. استفاده از ForAll به جای Foreach در زمان parallelization
   7. محدود کردن استفاده از OrderBy ها
5. Entity framework vs nhibernate vs dapper؟
6. Splitted query ها چیستند؟
7. از این قسمت به بعد مربوط به کورس EF از domtrain می شود.
8. ORM چی هست اصلا؟
   1. مخفف Object Relational Mapper هست که وظیفه اش تبدیل مدل ها، داده ها و متد ها به جداول،‌ رکورد ها و دستورات معادل در دیتابیس هست. با کمک ORM ما میتونیم بدون اینکه وابسته به دیتابیس خاصی باشیم، کار هایی که لازم داریم سمت دیتابیس انجام بشن رو داخل کد برنامه بنویسیم و بعدش خود ORM کار های مربوط به تبدیل کد به داده و تبدیل متد ها به دستورات دیتابیس رو انجام میده.
   2. از مزایای این ابزار اینه که یه بار کد رو می نویسیم و دیگه وابستگی به دیتابیس خاصی نداریم کما اینکه هر دیتابیس میتونه دستورات و ساختار خودشو داشته باشه اما ما دیگه برامون فرقی نمی کنه که دیتابیس چی باشه. همچنین دوباره نویسی کد ها و الزام به نیاز به یادگیری ساختار دیتابیس مورد نظر رو از بین می بره.
   3. یه مزیت دیگه اینه که طراحی دیتابیس و جداول و هندلینگ دیتا کاملا سمت کد انجام میشه و دیگه کاری با خود دیتابیس نداریم از طرفی هم نگرانی بابت اینکه اشکالی در ساختار دیتابیس و جداول به وجود بیاد نیستیم چون تمام قواعد و نکات رو از طریق کد داریم هندل می کنیم.
   4. مزیت بعدی قابلیت trace کردن کد هست؛ در حالتی که از ORM استفاده نکنیم،‌ معمولا دستورات سمت دیتابیس رو یا از طریق صدا کردن متدهای دیتابیسی (مثلا stored procedure ها در SQL) یا گذاشتن این دستورات در string literal ها اجرا می کنیم. مشکل اینجاس که این دستورات دیگه قابلیت trace و debug رو ندارن و در نتیجه در طولانی مدت به مشکل می خوریم اما با ORM، دیگه این مسائل به وجود نمیاد.
9. DbContext چیه؟
   1. عامل ارتباطی ما در EF بین کد C# و دیتابیس، ایشونه. ما به طور عادی برای اتصال بین دیتابیس و کد باید یه connection ایجاد کنیم که در ذات خودش یه  
      3-way handshaking داره. این عملیات میتونه هزینه بر باشه و هندل کردن هر باره اون هم دردسر. DbContext میاد و این connection رو هندل و تا هر زمان که نیاز باشه open نگه اش میداره. همچنین هر گونه عملیاتی که بخوایم روی دیتابیس انجام بدیم، این شئ هست که واسط اصلی بین دیتابیس و کد میشه و امکان انجام اون عملیات رو فراهم میکنه.
10. چرا گاهی این شبهه به وجود میاد که EF core سریع نیست؟
    1. کاری که DbContext انجام میده اینه که یه Mapping بین جداول دیتابیس و class model های داخل کد انجام میده. همچنین rule ها و در کل هر آنچه که لازمه تا ارتباط بین کد و دیتابیس برقرار کنه رو هندل میکنه. نکته ای که وجود داره اینه که تمام این کار ها رو در اولین call ای که روی DbContext اتفاق می افته انجام میده اما بعد از اون دیگه نیازی به تعریف مجدد این موارد از اول رو نداره و در نتیجه همیشه بار اولی که یه کد ران میشه،‌ این حس به وجود میاد که EF core داره کند عمل میکنه.
    2. ما هم که توسعه دهنده ایم،‌ معمولا اینطوریه که کد رو می نویسیم و بعدش یه بار ران میکنیم تا نتیجه رو ببینیم و طبق این گفته، کد که ران بشه بار اولی که DbContext رو call کنیم کند عمل میکنه اما در ادامه به اون سرعت و عملکرد مورد انتظار میرسه و طبق اون عمل میکنه.