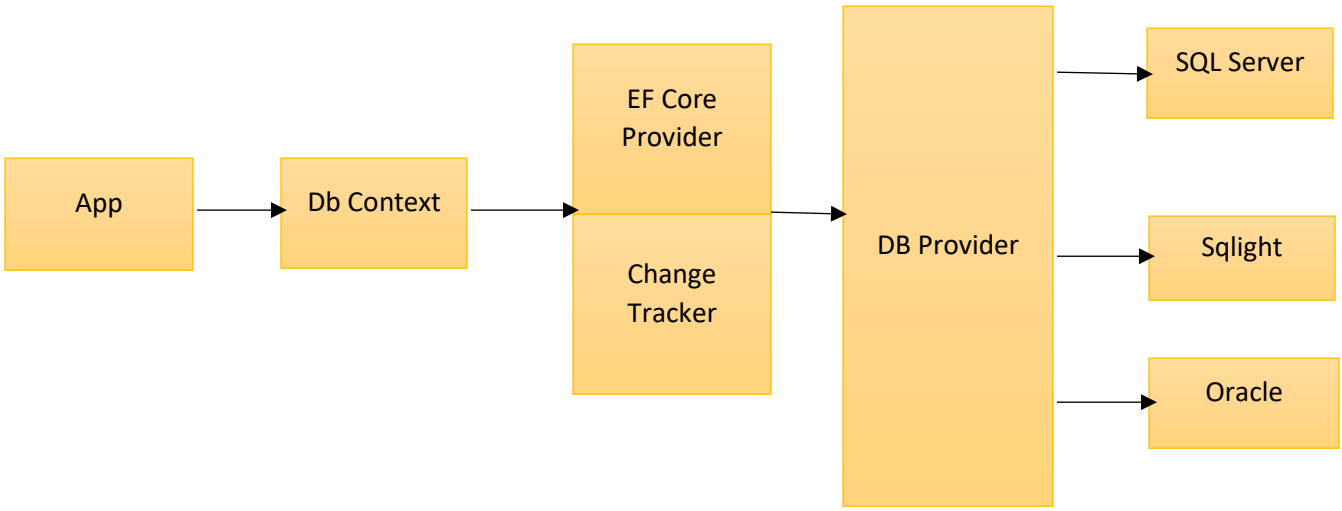
Entity Framework

@Hamid_Mohamadian

https://github.com/HamidMohammadi1990

جدول محتوا

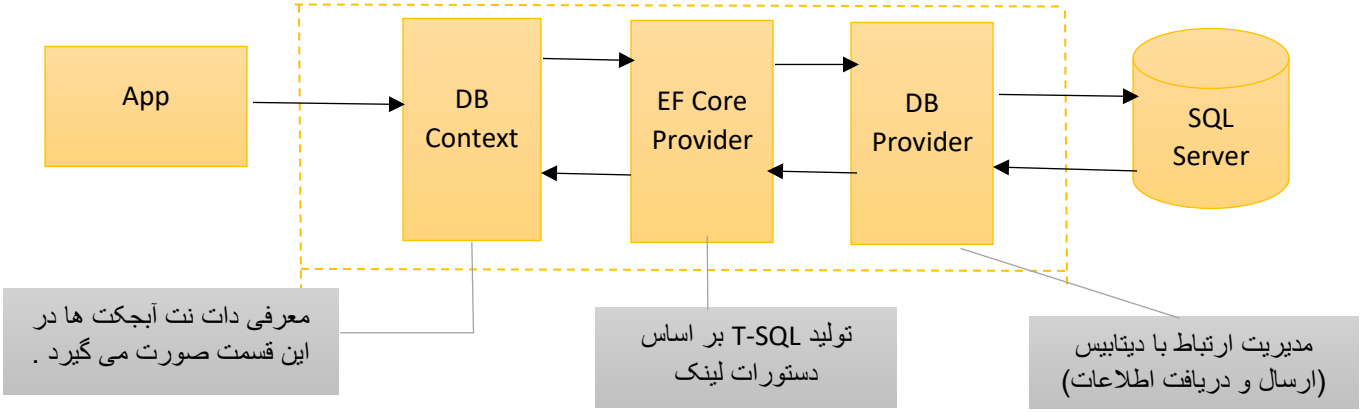
EF ORM



EF برای اینکه بتواند با دیتابیس های مختلف کار کند لایه مجزایی را برای مدیریت این موضوع در نظر گرفته به نام DB Provider که برای دیتابیس های مختلف DB Provider های مختلفی وجود دارد.

ما در پروژه یک سری کلاس هایی را برای داشتن جدول متناظر در دیتابیس ایجاد می کنیم ، EF از چه طریقی تشخیص می دهد که چه کلاس هایی را شناسایی کند برای ساختن جدول متناظر ؟

برای این موضوع یک Context وجود دارد که در آن .Net Object ها با DbSet معرفی شده اند تا EF جدول متناظر را برای ما بسازد .



معرفی پکیج EntityFrameworkCore.Design

این پکیج مسئول ایجاد Migration و Snapshot می باشد که فقط در محیط Development استفاده می شود .

پکیج فوق بایستی در پروژه Startup ما نصب باشد تا EF بتواند کارهای خود را به درستی انجام دهد .

نکته : این پکیج Development Dependency Package هست و فقط در زمان Develop مورد استفاده قرار می گیرد و در زمان Production نیازی به آن نداریم و هم چنین این پکیج Flow نمی خورد ، یعنی زمانی که این پکیج در لایه A نصب شده باشد و لایه A را به لایه B رفرنس بدهیم ، این پکیج انتقال داده نمی شود .

انواع **Command**

ما در EF دو نوع Command داریم

۱ - **Target Project**

مانند دستور Add-Migration یا دستور Install-Package که برای اجرای آن ها بایستی پروژه مورد نظرمان را انتخاب نماییم .

۲ - **Startup Project**

مانند دستور Update-Database که برای اجرای آن حتما بایستی Startup Project انتخاب شده باشد .

نکته : DbContext در واقع EntryPoint ما می باشد ، چون برای هر کاری نیاز هست که ما یک نمونه از Context داشته باشیم.

نکته : طول عمر DbContext بایستی چقدر باشد ؟

طول عمر DbContext بایستی به اندازه ی باز کردن Session در دیتابیس ، دریافت اطلاعات و بستن Session مربوطه باشد و

در کل طول عمر DbContext بایستی کوتاه باشد.

تعریف Unit of work

شامل تمامی مواردی که در درون یک Business Transaction اتفاق می افتد می باشد و این Business Transaction می تواند یک تغییر و یا دریافت اطلاعات از دیتابیس باشد.

روش های **Config** کردن **Context**

ما از سه طریق می توانیم Context را Config کنیم .

۱ - **Override** کردن متد **OnConfiguring** در **Context** پروژه

۲ – دریافت آن در متد سازنده در زمان نمونه سازی (از طریق نمونه سازی DbContextOptionBuilder)

۳ - زمانی که نمونه سازی **Context** را به یک **IOC Container** می سپاریم و مقدار دهی Option را در زمان اجرا

هم به خود IOC Container می سپاریم ، مانند Services.AddDbContext موجود در پروژه های Net. که در Program پروژه آن را Config می کنیم .

ما وقتی Add-Migration را اجرا می کنیم EF از چه طریقی متوجه می شود که چه Property هایی کم و یا اضافه شده و چه تغییرات دیگری صورت گرفته است ؟

به صورت کلی ابتدا بایستی در زمان Design یک نمونه از دیتابیس ساخته شود و نمونه ساخته شده را با Snapshot موجود مقایسه کند تا متوجه تغییرات اعمال شده شود.

نکته اینجا مطرح می شود که این نمونه سازی به چه صورت انجام می گیرد ؟

در EF مدیریت این مکانیزم به دست کلاس DbContextOperation سپرده شده که یک نمونه از دیتابیس می سازد و برای شناسایی Context پروژه سه مورد زیر را چک می کند .

1 - آن DbContext هایی که اینترفیس IDesignTimeDbContextFactory را پیاده سازی کرده اند .

2 - Service Provider را بررسی می کند ، که در واقع مربوط می شود به Context معرفی شده در IServiceCollection که در Program آن را معرفی می کنیم .

3 - در این مرحله به دنبال کلاسی می گردد که از کلاس DbContect ارث بری کرده باشد.

نکته : اگر بخواهیم تنظیمات Context در Design Time از Execution Time متفاوت باشد ، بایستی در لایه ای که در آن Context وجود دارد ، یک کلاس مربوطه ساخت و از اینترفیس IDesignTimeDbContextFactory ارث بری کرد و در متد CreateDbContext یک نمونه از دیتابیس با تنظیمات دلخواه را برگردانیم .

نکته : ما هنگام استفاده از پکیج EF یک کلاس داریم به نام DbContext که متعلق به EF می باشد و یک AppContext ایجاد می کنیم و از کلاس DbContext ارث بری می کنیم .

سوال پیش میاد که آیا ارث بری از کلاس AppContext صورت خواهد گرفت ؟

خب بدیهی هست که این ارث بری هیچ وقت صورت نخواهد گرفت و برای جلوگیری از ارث بری بایستی AppContext را Seald در نظر گرفت .

بایستی اشاره کنم که این موضوع شامل Best Practice ها نیست و فقط چارچوب را مشخص می کند که دیگر ارث بری صورت نخواهد گرفت.

روش های معرفی **Entity** به **Context**

برای انجام این کار سه روش زیر وجود دارد

۱ - معرفی به واسطه ی **DbSet** در **Context**

```
Public DbSet<User> Users { get; set; }
```

۲ - معرفی **Entity** در متد **OnModelCreating**

در Context پروژه یک متد وجود دارد به نام OnModelCreating که متعلق به EF می باشد و بایستی Entity مربوطه را به روش زیر معرفی کرد .

```
public override void OnModelCreating(ModelBuilder  builder)
{
   builder.Entity<User>();
}
```

۳ – استفاده از یک **Entity** در **Entity** دیگر به عنوان **Navigation property**

در مرحله آخر اگر یک Entity در هیچ جایی به عنوان یک Entity معرفی نشده بود ، اما اگر در یک Entity دیگری به عنوان Navigation Property استفاده شده بود ، EF آن را هم به عنوان یک Entity شناسایی می کند و جدول متناظر در دیتابیس ساخته می شود .

اولویت EF برای در نظر گرفتن تنظیمات برای Entity و Property های مربوطه اش به صورت زیر می باشد.

ابتدا Convention های پیش فرض خود را در نظر می گیرد و اگر از Data Annotation استفاده کرده باشیم ، Data Annotation را در اولویت قرار می دهد اما اگر از Fluent Api استفاده کرده باشیم آن را در اولویت قرار می دهد .

پس Fluent Api در اولویت اصلی می باشد و نسبت به Convention های پیش فرض و Data Annotation ارجعیت بالاتری دارد.

هنگامی که از **Fluent Api** استفاده می کنیم ، در سه سطح زیر می توانیم تنظیمات را اعمال نماییم.

۱ - Mode Wide Configuration

کانفیگ کردن در سطح تمامی Entity ها ، به طور مثال تنظیم کردن طول تمامی Property هایی که String هستند و یا تنظیم کردن Schema تمامی Entity ها .

```
builder.HasDefaultSchema("dbo");
```

۲ - Type Configuration

کانفیگ کردن در سطح یک Entity

```
builder.Entity<User>().ToTable("People");
```

۳ - Property Configuration

کانفیگ کردن در سطح یک Property متعلق به یک Entity , به طور مثال

```
builder.Entity<User>()
.Property(x => x.Name)
.HasMaxLength(50);
```

بحث Loading Types

۱ - Eager Loading

```
var userWithOrders =
    context.User
    .Single()
    .Include(x => x.Orders);
```

۲ – Explicit Loading

```
var use = context.Users.Single();
context.Entry(user).Collection(x => x.Orders).Load();
```

۳ - Lazy Loading

اگر بخواهیم در زمان واکشی یک یا چندین رکورد دیتاهای وابسته به آن نیز به صورت خودکار واکشی شود از این ویژگی استفاده می کنیم .

برای استفاده بایستی ابتدا پکیج EntityFrameworkCore.Proxies را نصب و در زمان رجسیتر کردن Context از UseLazyLoadingProxies() استفاده کرد .

نکته : باید در نظر داشت که برای اجرا شدن صحیح این ویژگی بایستی Navigation Property ها را Virtual در نظر گرفت .

```
context.Users.Where(x => x.Nzme == "ali").ToList();
```

Entry Point Linq Query Execution Command

نکته : در کوئری بالا چون ما داریم با منبعی Out Source کار می کنیم ، قسمت Where را به یک Expression Tree تبدیل می کند و بعدا EF این Expression Tree را تبدیل به T Sql می کند . اما اگر منبع در Memory موجود بود ، قسمت Where تبدیل به Func می شد .

نکته : اگر بخواهیم مایگریشن هایی که وضعیت Pending دارند را واکشی کنیم ، از دستور زیر استفاده می کنیم.

```
var pendingMigrations = context.Database.GetPendingMigrations();
```

نکته : اگر بخواهیم در فضای RunTime به آخرین مایگریشن Migrate کنیم از دستور زیر استفاده می کنیم .

```
context.Database.Migrate();
```

بحث Client And Server Evaluation

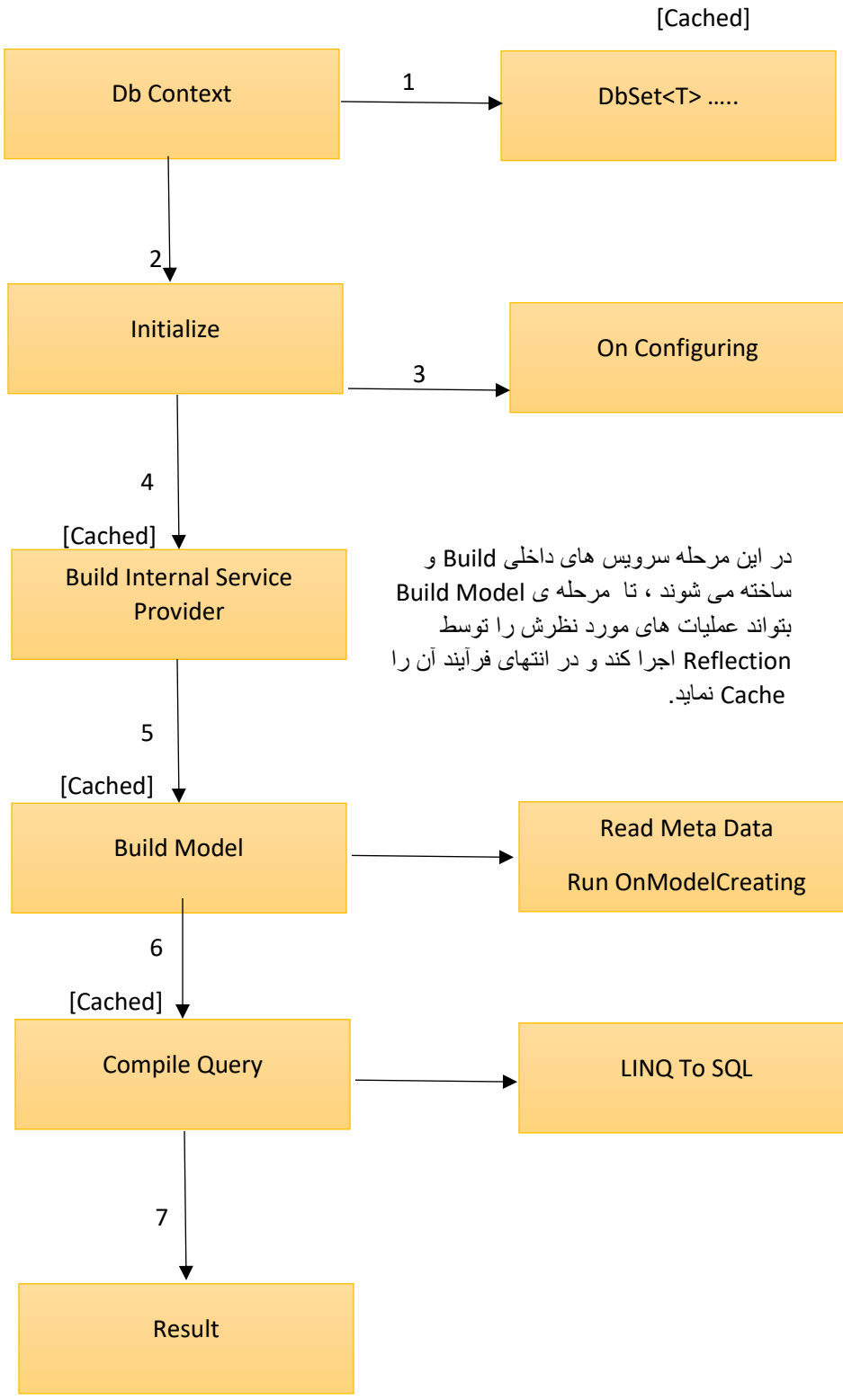
```
var result =
context.Users
.Where(x => x.Name == "ali")
.Select(x => new
{
    FullName = Combine
    (
        x.Name, x.LastName
    )
})
.ToList();
```

تا این قسمت که Whereهست ، Server Side می باشد ، یعنی دیتا سمت دیتابیس فیلتر می شود.

این قسمت Client Side می باشد و دیتا سمت نرم افزار Combine می شود و Select را اجرا می کند.

.

Steps For Query Execution In EF تشریح



نکات :

1 - اگر یک نمونه از Context داشته باشیم و مجددا بیاییم و یک نمونه دیگری از Context بسازیم ، دیگر مرحله اول که OnModelCreating هست اجرا نمی گردد چون در نمونه سازی اولیه از Context این مرحله Cache شده است .

2 - هنگامی که نمونه دوم از Context ساخته می شود ، دیگر مرحله چهارم یعنی Build Internal Services اجرا نمی گردد ، چون در نمونه سازی اولیه از Context این مرحله Cache شده است .

3 - کوئری که کامپایل می شوند Cache می شوند تا از کامپایل کردن مجدد آن ها جلوگیری شود .

4 - مرحله سوم یعنی OnConfiguring نمی تواند Cache شود ، چون این امکان وجود دارد که Instance اول با Instance دوم Config های مختلفی داشته باشد . به همین خاطر این مرحله امکان Cache شدن ندارد.

نکته : معماری EF از نوع Service Based Architecture هست .

اگر بخواهیم در یک کوئری بر روی یک Shaddow Property کوئری بزنیم و یا شرط در نظر بگیریم بایستی به صورت زیر عمل کنیم :

```
context.Users
.Where(x => EF.Property<int>(x, “Propertyname”))
.ToList();
```

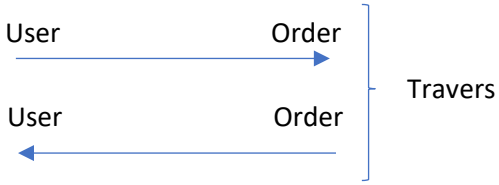
قابلیت NRT (Null Refrence Type)

به طور پیش فرض Value Type ها Nullable نیستند و آن ها را Required در نظر می گیرد، اما Refrence Type ها به صورت پیش فرض Nullable هستند. اگر بخواهیم این Convention را تغییر بدهیم تا Refrence Type ها را Required در نظر بگیرد ، بایستی در کانفیگ خود پروژه کانفیگ زیر را قرار بدهیم .

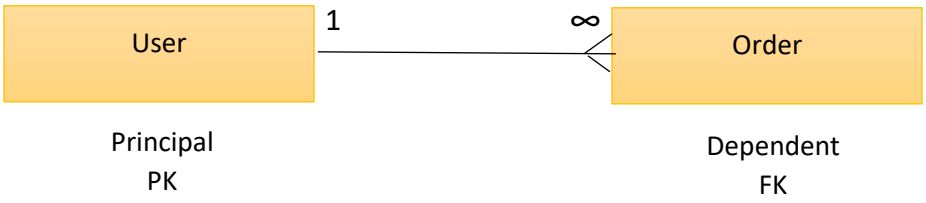
```
<nullable>enable</nullable>
```

تعریف روابط

```
builder
.HasMany(x => x.Address)
.HasOne(x => x.User)
.HasForeignKey(x => x.User)
.HasPrincipalKey(x => x.Id);
```



نکته : به هر یک از این رفت و برگشت ها که تشکیل یک رابطه را می دهند ، Travers می گویند.



نکته : امکان دارد از User به Order رابطه برقرار شود ، اما بلعکس آن صورت نگیرد که به این نوع رابطه UniDirectional Relation می گویند.

نکته : زمانی که از UniDirectional Relation استفاده می کنیم ، چون آن Navigation قابلیت Nullable بودن را دارد ، در زمان کوئری گرفتن کامپایلر به ما Warning می دهد که این Property می تواند Null باشد ، اما خود EF با این موضوع مشکلی ندارد .

برای درک بیشتر به مثال زیر دقت کنید .

```
var orders =
context.Orders
.Where(x => x.User.FirstName == "ali").ToList();
```

در کوئری بالا چون User می تواند Null باشد به همین خاطر کامپایلر به ما Warning می دهد که برای رفع این Warning بایستی از قابلیت Null Forgiving Operator استفاده کرد .

پس کوئری بالا را می توانیم به شکل زیر بنویسیم.

```
var orders =
context.Orders
.Where(x => x.User!.FirstName == "ali").ToList();
```

بحث **OwnsOne**

```
class User
{
   Public Address Address { get;  set; }
}

[Owned]
class Address  { }
```

در تعریف بالا ما یک رابطه یک به یک داریم ، اما اگر بخواهیم بگوییم که درست هست که کلاس Address جدا هست و Property هایش داخل کلاس Uaer نیست و فقط خود کلاس Address استفاده شده ، اما تو بیا کلاس Address را یک جدول جدا در نظر نگیر و آن را زیر مجموعه کلاس User در نظر بگیر ، در این صورت به شکل زیر عمل می کنیم :

با اتربیوت [Owned] می توانیم بر روی کلاس Address , زیر مجموعه بودن آن را مشخص کنیم و معادل این اتربیوت در Fluent Api به صورت زیر می باشد.

```
builder.OwnsOne(x => x.Address);
```

بحث **OwnsMany**

```
class User
{
    public List<Tag> Tags { get; set; }
}
class Tag {}
```

اگر در تنظیمات کلاس User کانفیگ زیر را داشته باشیم

```
builder .OwnsMany(x => x.Tags);
```

تفاوت دو شکل بالا در چیست ؟

```
var user = context.User.ToList();
```

کوئری فوق فقط کاربران را واکشی می کند ، اما اگر بخواهیم سفارش های کاربران را نیز واکشی کند بایستی کوئری فوق را به شکل زیر بنویسیم .

```
var user = context.User.Include(x => x.Orders).ToList();
```

اما در رابطه با User و Tags اگر کوئری زیر اجرا شود تمامی Tag ها را به همراه کاربران واکشی می کند ، چون کلاس User مالک کلاس Tag می باشد .

```
var user = context.Users.ToList();
```

به کلاس هایی که Owned Entity Type هستند Key Less Entity نیز می گویند.

نکته : در بین دو جدول User و Tag که جدول User مالک و جدول Tag ، Owned هست چندین نکته به شرح زیر وجود دارد :

```
var user = context.Tags.ToList;()
```

1 - در اینجا ما نمی توانیم لیست Tag ها را دریافت کنیم ، چون زیر مجموعه User می باشد و اگر مستقیما بخواهیم لیست Tag ها را واکشی کنیم با خطا مواجه می شویم .

2 - ما نمی توانیم Tag را به صورت DbSet در Context پروژه رجیستر کنیم و اگر این کار را انجام بدهیم در زمان RunTime با خطا مواجه می شویم .

نکته : همانطور که Query ها در EF به سه بخش تقسیم می شوند ، Command ها هم از بخش زیر تشکیل می شوند :

```
Context.Users.Add(user);
```

بخش دوم بخش اول

```
Context.SaveChanges();
```

بخش سوم

بحث Entity States

در EF هر نمونه ساخته شده از هر Entity از دیدگاه Change Tracker می تواند وضعیت های مختلفی داشته باشد که در ادامه به آن ها می پردازیم.

۱ - Added

به این معنی می باشد این نمونه در سمت EF موجود می باشد اما در دیتابیس موجود نمی باشد و بایستی در SaveChange در دیتابیس Insert صورت گیرد.

۲ - UnChanged

به این معنی می باشد که این نمونه هیچ تغییری نکرده است و همانند آن در دیتابیس موجود می باشد.

۳ - Modified

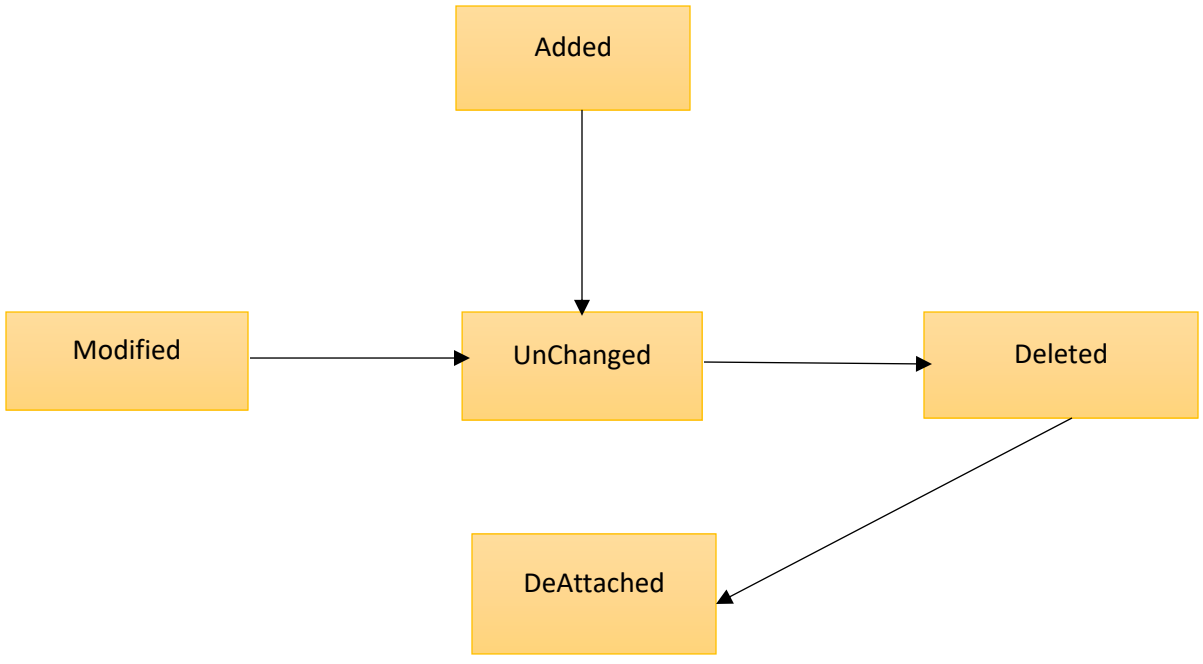
به این معنی می باشد که یک الی چندین از پراپرتی های نمونه تغییر کرده و بایستی در دیتابیس آپدیت صورت گیرد.

۴ - Deleted

نمونه هایی که این وضعیت را دارند EF کوئری حذف برای آن ها می سازد و بایستی از دیتابیس حذف گردند.

۵ - Detached

نمونه هایی که وضعیت Deleted دارند بعد از فراخوانی متد SaveChanges وضعیت آن به Detached تغییر می کند ، هم چنین وقتی یک نمونه از یک Entity می سازیم وضعیت اولیه آن Detached می باشد.



تفاوت Entity و Entry

Entity

آن چیزی هست که EF مطابق با آن یک جدول در دیتابیس می سازد.

Entry

در واقع متدی است که اگر بخواهیم با ChangeTracker کار کنیم ، یک سری متدها را برای تغییرات بر روی Entity هایی که Track شده اند در اختیارمان می گذارد.

نکته : یکی از استراتژی های ChangeTracker برای Track کردن نمونه هایی از Entity ها که پیش فرض هم هست ، استراتژی SnapShot می باشد .

نکته : متد SaveChanges در ابتدای کار برای متوجه شدن تغییرات متد Detect Changes را فراخوانی می کند.

نکته : Life Time یک Context به ازای هر درخواست می باشد (Scope)

بحث Relation Fixup

```
var users =  context.Users.ToList();
var orders =  context.Orders.ToList();
```

در کوئری اولی user ها را واکشی و Track می کند و اگر context.ChangeTracker.DebugView.LongView را اجرا کنیم ، مشاهده می کنیم که Order های کاربران null هستند .

در کوئری دوم سفارشات واکشی و Track می شوند و بعد از واکشی سفارشات Relation Fixup اتفاق می افتد و EF با بررسی Foreign Key های آن بررسی می کند که چه کاربرانی رو Track کرده بوده و در صورتی که کاربری را پیدا کند که این Order متعلق به آن باشد این دو نمونه را با ساختاری به یکدیگر وصل می کند .

در ادامه اگر مجددا context.ChangeTracker.DebugView.LongView را فراخوانی کنیم متوجه می شویم که دیگر Order های کاربرانی که Order برای آن ها ثبت شده بوده دیگر Null نیست و EF آمده آیدی سفارش های مربوط به کاربر رو با ساختار زیر قرار داده است .

Orders : [{4 ,3 ,2 ,1}]

به این فرآیند Relation Fixup می گویند.

نکته : در کوئری های قبلی که کاربران و سفارشات را جدا جدا واکشی کردیم ، اگر به صورت زیر هم آن ها را یکجا واکشی کنیم مجددا Relation Fixup رخ می دهد.

```
var users =
context.Users
.Include(x => x.Orders)
.ToList();
```

سوال : چرا **Relation Fixup** برای **EF** مهم هست ؟

چون EF این قابلیت را به می دهد تا از طریق Navigation Property ها بتوانیم عملیات Add یا Remove داشته باشیم.

نکته : در سطح دیتابیس برای یک رابطه PK و FK کافی می باشد ، اما در EF به جز این دو نیاز به Navigation Property نیز دارد و به همین خاطر فرآیند Relation Fixup رخ می دهد.

بحث **Delete OrphansTiming** یا اصطلاحا **Reparrenting**

سناریو از این قرار است که ما دو تا لیست داریم که کاربران به همراه سفارشاتشان در آن ها قرار دارند و ما می خواهیم یک یا چند سفارش را از کاربر اولی حذف و به کاربر دومی که در لیست دوم موجود هست اضافه نماییم .
برای درک بهتر مطلب به کد زیر توجه نمایید تا ادامه توضیحات را بعد از آن داشته باشیم.

```
var user1 =
context.Users
.Include(x => x.Orders)
.FirstOrDefault(x => x.Id == 4);

var user2 =
context.Users
.Include(x => x.Orders)
.FirstOrDefault(x => x.Id == 5);

var orderForUser2 = user1.Orders.First();
user2.Ordets.Remove(orderForUser2);
```

در مرحله فوق orderForUser2 وضعیت Deleted را دارد.

```
user1.Orders.Add(orderForUser2);
```

اما در این مرحله چون orderForUser2 را به یک کاربر دیگری نسبت دادیم ، وضعیت Deleted به Modified تغییر پیدا می کند ، اما یک نکته ای وجود دارد که در ادامه به آن اشاره خواهیم کرد.

```
context.SaveChanges();
```

ما در کد فوق ابتدا آمدیم و یک سفارش دلخواه از کاربر دوم را حذف و به کاربر اولی اضافه نمودیم ، اما در این سناریو EF تمامی Property های آن را Modified قرار می دهد ، در صورتی که ما فقط UserId آن را تغییر دادیم ، برای رفع این موضوع بایستی از قابلیت OrPhansTiming استفاده شود که در ادامه به آن می پردازیم.
در زمان استفاده از قابلیت OrPhansTiming در سناریو قبلی به این صورت می باشد که زمانی که سفارش از کاربر اولی حذف شد ، سریعا وضعیت آن سفارش را به Deleted تغییر نده و صبر کن تا شاید تا قبل از فراخوانی متد SaveChanges آن سفارش به کاربر دیگری اضفه شد و در صورت اضافه شدن سفارش به کاربر دیگر ، وضعیت آن به Modified تغییر پیدا می کند.

برای استفاده از قابلیت OrPhansTiming می توانیم از کد زیر استفاده کنیم.

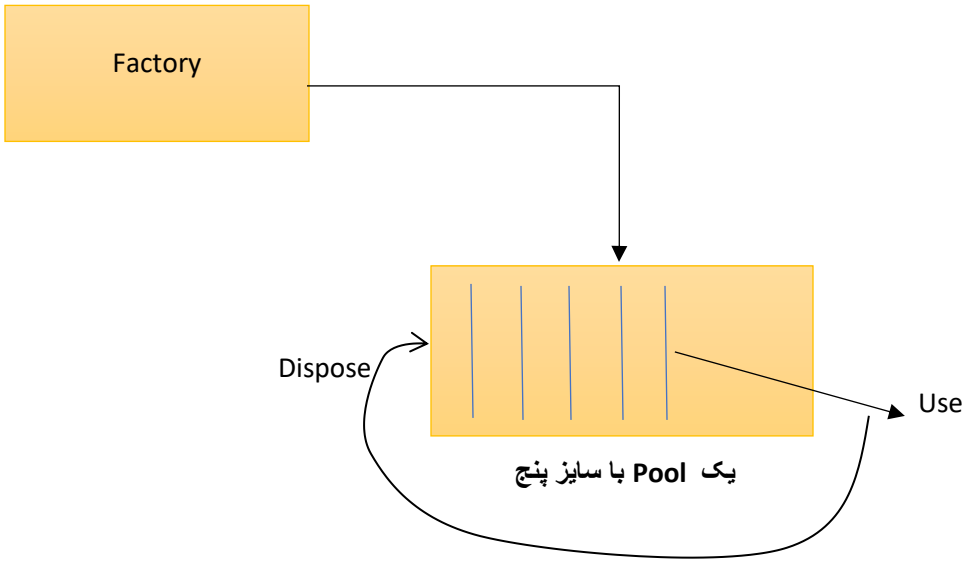
```
context.ChangeTracker.DeleteOrPhansTiming = CascadeTiming.OnSaveChanges;
```

پس زمانی که از این قابلیت استفاده می کنیم ، زمانی که سفارش از کاربر اولی حذف شد وضعیت آن سفارش به Modified تغییر می کند و هم چنین UserId نیز Null می شود . (در واقع با Null شدن UserId آن سفارش Orphan و یا یتیم می شود)
در نهایت در زمان فراخوانی متد SaveChanges اگر آن سفارش Orphan بود ، آن سفارش پاک خواهد شد و در غیر اینصورت UserId آن کاربر ویرایش خواهد شد.

نکته : زمانی که سفارش از کاربر دومی حذف و به کاربر اولی اضافه شد ، اصطلاحا Reparrenting صورت گرفت مجددا Relation Fixup اتفاق می افتد.

بحث DbContext Pool

نکته : سایز پیش فرض Context Pool 1024 می باشد.



در زمان ساخت ، که ما به یک نمونه از Context نیاز داریم ، Factory بالا بر چه اساسی آن نمونه مربوطه را به ما بر می گرداند ؟! (منظور این است که اولین یا دومین یا ... نمونه را بر می گرداند ؟!)

این Factory بر اساس الگوریتم FIFO عمل می کند و بعد از انتخاب یک نمونه و یا Dispose شدن آن ، نمونه را به ابتدای صف انتقال می دهد.

نکته : برای نگهداری این نمونه هایی از Context از ساختمان داده Queue استفاده می کند که قابلیت FIFO را دارد.

نکته : این Pool که یک Queue هست و قابلیت FIFO را دارد ، آیا این ویژگی ها کافی هستند یا بایستی ویژگی ای اضافه و یا ساختاری تغییر کند ؟

بله بایستی ساختار تغییر کند و امکان پیچیدگی بیشتر هست ، به طور مثال اگر دو Thread هم زمان درخواست Context کنند ، یک نمونه به دو درخواست داده می شود ، پس باید Thread Safe نیز باشد . پس بایستی به جای استفاده از Queue می بایست از ConCurrent Queue استفاده کرد .

سوال) به فرض که سایز Context Pool ما 10 می باشد . اگر این 10 تا نمونه استفاده شوند و نمونه آزادی نداشته باشد و ما درخواست Context یازدهمی را کنیم ، چه اتفاقی می افتد ؟

چون نمونه آزاد ندارد ، می بایست به صورت نرمال یک نمونه بسازد و به صورت نرمال هم آن را Dispose کند.

نکته : در این Pool الگویی که برای هر کانتکست استفاده می کند ، الگوی Singleton می باشد ، چون در زمان Dispose شدن فقط آن Context می بایست Reset State شود.

نکته : یکی از محدودیت های Context Pooling این هست که متدهای OnModelCreating و OnConfiguring تنها یک بار اجرا می شوند و در زمان ساخت نمونه های بعدی دیگر این دو متد اجرا نمی شوند.

نکته : آیا زمانی که ما یک Context Pool با سایز 1024 داریم ، در ابتدای کار 1024 نمونه وجود دارد ؟

خیر ، به مرور زمان و با هر بار Dispoae شدن Context به این Pool اضافه می شود.

بحث TVF و View

نکته : ما در دیتابیس مواردی داریم تحت View , Store Procedure , Table Value Function (TVF) . ما چطور می توانیم در EF با این موارد کار کنیم ؟

در اینجا EF یک امکانی را فراهم کرده تحت عنوان Keyless Entity و بایستی به صورت DbSet به Context معرفی شود و چون PK ندارند دیگر ChangeTracker آن ها را Track نمی کند.

برای استفاده از View در EF بایستی مراحل زیر را طی کنیم .

1 - ابتدا View مورد نظر در دیتابیس ساخته می شود.

2 - یک Keyless Emtity با پراپرتی هایی که مطابق با نتیجه View هستند می سازیم و به صورت DbSet در Context معرفی می شود.

3 - در متد OnModelCreating بایستی به صورت زیر Keyless Entity خود را Config نماییم.

```
builder.Entity<User_View>()
.HasNoKey()
.ToView("User_View_On_Db", "dbo");
```

برای استفاده از TVF ها همانند View می توانیم رفتار کنیم و برای استفاده از آن ها به صورت زیر عمل می کنیم .

1 - ابتدا بایستی یک TVF در دیتابیس ایجاد گردد.

2 - یک keyless Entity می سازیم و به صورت DbSet آن را به Context معرفی می کنیم .

3 - در متد OnModelCreating بایستی به صورت زیر Keyless Entity خود را Config نماییم .

```
builder.Entity<User_TVF>()
.HasNoKey()
.ToFunction("User_TVF_On_Db");
```

نکته : در زمان استفاده از View یا TVF ما می توانیم در زمان کوئری گرفتن از Where استفاده کنیم و شروط ما سمت دیتابیس اعمال شوند (به صورت Server Side).

بحث Interceptor

اینترسپتورها یک سری نقاط مشتص شده در Pipline خود EF می باشد که به ما اجازه می دهند قبل و یا بعد از یک سری رویدادها ، یک سری کارها را انجام دهیم . به طور مثال لاگ زدن قبل و یا بعد از فراخوانی متد SaveChanges .

نکته : انواع مختلفی Interceptor وجود دارد مانند , SavingChangesInterceptor , DbCommandInterceptor MaterializationInterceptor که تمامی آن ها از IInterceptor ارث بری کرده اند.

نکته : درصد بالایی از Interceptor ها Stateless هستند و قرار نیست وضعیتی را نگهداری کنند.

نکته : همیشه Interceptorها Stateless نیستند و می توانند StateFull هم باشند و وضعیتی را نگه دارند . در اینجا ما بایستی حواسمان به یک چیز باشد و آن هم بحث هم زمانی هست که نبایستی Lock Object را فراموش کنیم .

بحث DbCommand

زمانی که EF Core Provider در حال تبدیل و یا کامپایل Linq به Sql هست ، نوع خروجی از نوع DbCommand هست.

نکته : هر چیزی که بین EF Core Provider و Db Provider قرار هست که ارسال گردد از نوع DbCommand می باشد.

نکته : ما چه کوئری را به صورت Linq بنویسیم (context.Users.ToList()) و یا چه به صورت context.Database.SqlQuery و چه به صورت ExecuteSqlRaw ، در نهایت اینترسپتورهای DbCommand اجرا خواهند شد .

نکته : کوئری ها به دو دسته زیر تقسیم می شوند :

1 - Query

مانند Select × From Users که N رکورد را به عنوان خروجی بر می گرداند.

2 - NonQuery

مانند اجرا کردن دستور Delete و یا Update که ما انتظار خروجی نداریم و فقط اعلام می کند که چه تعداد رکوردی حذف و یا ویرایش شده است .

بحث **Materialization**

زمانی که یک کوئری به شکل زیر داریم

```
context.Users.ToList();
```

به طول مثال اگر ده تا کاربر وجود داشته باشد ، زمانی که کوئری بالا اجرا می شود و دیتاها را با Reader از دیتابیس می خواند و از آنجایی که EF یک ORM هست و ما انتظار داریم که EF دیتاهایی که از دیتابیس خوانده است را به کلاس User اصطلاحا Map کند .

در این فاز دو مرحله زیر بایستی صورت گیرد :

1 - Create Instance Of User

ابتدا برای مقدار دهی بایستی یک نمونه از Entity داشته باشیم که در مرحله اول صورت می گیرد .

2 - Initialize Data

در این مرحله نمونه ساخته شده در مرحله اول مقدار دهی می شود .

این دو مرحله بایستی به تعداد نتایجی که از دیتابیس دریافت کرده است تکرار شود . به این دو فاز اصطلاحا Materializatiin می گویند .

```
var context1 = new AppContext();
var context2 = new AppContext();
```

در زمان ساخت هر Context و هنکاهی که می خواهیم از آن نمونه استفاده کنیم ، متد OnConfiguring اجرا می گردد ، جایی که اینترسپتورهای ما رجیستر شده اند و هر دفعه بایستی یک نمونه از آن اینترسپتور ساخته شود.

در اینجا یک سوالی مطرح می شود ؟! مگر ما نگفتیم که اینترسپتورها Stateless هستند و قرار نیست وضعیتی را نگه دارند ؟ خب در اینجا چرا بایستی هر دفعه در متد OnConfiguring یک نمونه جدید ساخته شود ؟

در اینجا ما می توانیم یک Static Property در قسمت بالای متد OnConfiguring بسازیم و همان را در OnConfiguring رجیستر کنیم .

بحث Soft Delete

ما دو نوع Delete داریم

۱ - Hard Delete

حذف به طور فیزیکی و کامل را گویند که دیتا از روی دیسک پاک می شود.

۲ - Soft Delete

در این نوع ما یک فیلدی تحت عنوان IsActive یا IsDelete داریم که با توجه به مقدار آن متوجه می شویم آن رکورد حذف شده است یا خیر .

اگر Entity ما قابلیت Soft Delete داشته باشد که برای حذف آن می بایست فقط IsActive آن را False قرار دهیم ، اما اگر یک برنامه نویس از روش Remove استفاده کرد چه اتفاقی می افتد ؟ چطور می توانیم جلوی این کار را بگیریم ؟

چون در اینصورت اطلاعات ما به صورت فیزیکی پاک می شوند ، در صورتی که Entity ما قابلیت Soft Delete دارد.

در این سناریو ما بایستی از Interceptor ها استفاده کنیم .

در ادامه می خواهیم برای رفع مشکل از SaveChangesInterceptor استفاده کنیم .

```
public class SoftDeleteInterceptor : SaveChangesInterceptor
{
    public override InterceptionResult<int> SavingChanges(DbContextEventData eventData,
        InterceptionResult<int> result(
        {
            if (eventData.Context is null)
                return result;

            foreach(var entry in eventData.Context.ChangeTracker.Entries())
            {
                if (entry is { State: EntityState.Deleted ,Entity: ISoftDelete entity })
                {
                    entity.IsActive =  false;
                   entity.State =  EntityState.Modified;
                }
            }
        }
 }
```

توضیحات اینترسپتور SoftDeleteInterceptor

خب ما در اینجا یک Interceptor نوشتیم تا قبل از اینکه متد SaveChanges اجرا شود یک سری تغییرات اعمال کند .

در اینجا میاد و روی تمامی Entity هایی که Track شده اند حلقه می زند و در صورتی که State آن Delete بود و از اینترفیس ISoftDelete ارث بری کرده بود ، بیاد و وضعیت آن را از Deleted به Modified تغییر دهد و برای اینکه رکورد دیگر نمایش داده نشود IsActive آن را False قرار می دهیم.

با این کار دیگر Hard Delete اتفاق نمی افتد ، حتی اگر برنامه نویس حواسش به این موضوع نباشد و بخواهد Hard Delete انجام دهد.

در اینترسپتورها دو کار زیر را می توانیم انجام دهیم

1 – Modify

می توانیم روند و یا دیتاهای مربوطه را تغییر دهیم .

2 – Suppress

می توانیم مسیر پایپ لاین اجرایی اینترسپتورها را تغییر دهیم و زمانی که به اینترسپتور ما رسید ، دیگر ادامه ندهد و Exit کند .

نکته : یکی از مشکلاتی که Soft Delete به ارمغان می آورد این هست که ما اگر دز آن Entity یک فیلد یا چند فیلد را Index و هم چنین Unique کرده باشیم ، فیلد یا فیلدهای مربوطه در دراز مدت زیاد می شوند و حجم ایندکس ما را افزایش می دهند و هم چنین آن فیلد تا زمانی که IsActive آن False هست رزرو باقی می ماند .

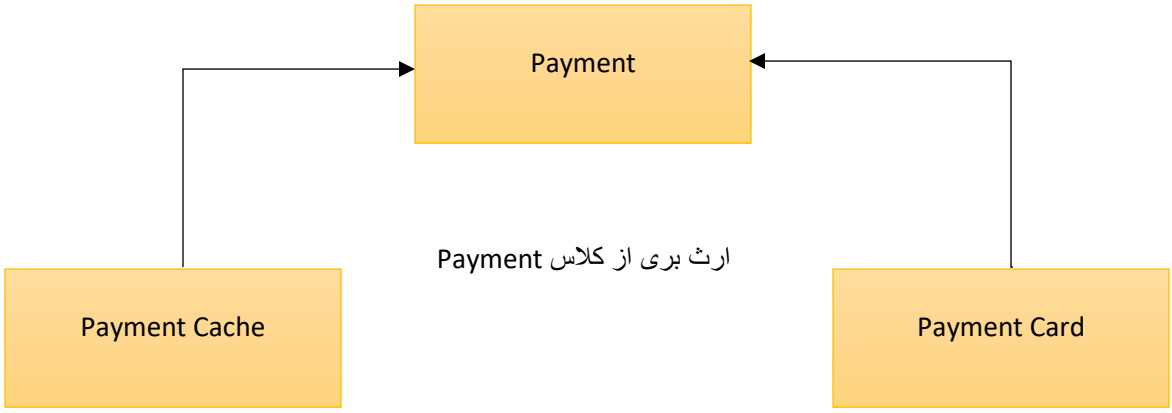
برای رفع این مشکل بایستی از Filter ها استفاده کنیم .

```
builder.HasIndex(x => x.UserName)
.IsUnique()
.HasFilter("[UserName] Is Not Null And [IsActive] = 1");
```

مبحث ارث بری

ما در EF ارث بری های مختلفی داریم که در ادامه هر یک را توضیح خواهیم داد.

۱ - TPH (Table Per Hierarchy)



به صورت پیش فرض EF کاری با Base Class ندارد و تنها کاری می کند این هست که پراپرتی های کلاس Base را هم در نظر می گیرد .

نکات :

- اگر ما Payment Cache را به Context معرفی کنیم ، جدول Payment Cache را به همراه پراپرتی های Payment می سازد.

- اگر ما Payment و هم Payment Cache را به Context معرفی کنیم ، در این حالت TPH اتفاق می افتد و برای این دو کلاس یک جدول را در نظر می گیرد .

- اگر ما هم Payment و Payment Cache و Payment Card را به Context معرفی کنیم ، برای هر سه کلاس یک جدول در نظر می گیرد.

در TPH برای اینکه EF بتواند تشخیص دهد که این رکورد برای کدام کلاس می باشد ، از یک Shaddow Property به نام Discriminator استفاده می کند.

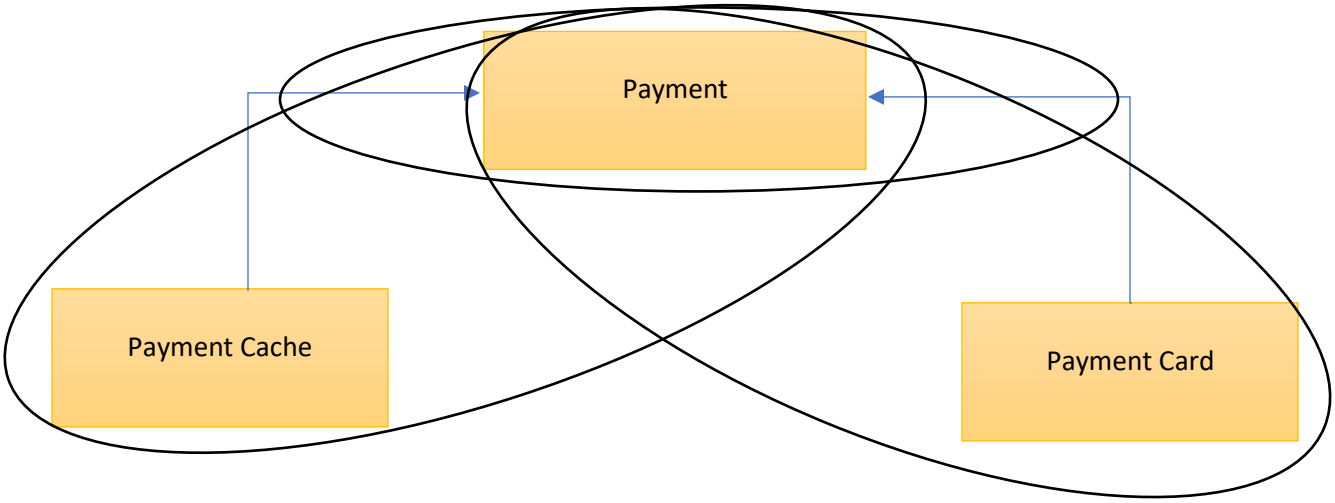
در TPH یک سناریو داریم که در Context کلاس پدر معرفی نمی شود و دو Entity که از کلاس پدر ارث بری کرده اند معرفی می شوند ، در این حالت EF دو جدول مجزا در نظر می گیرد و پراپرتی های پدر را در دو جدول فرزند قرار می دهد .

مشکل اینجا هست که ما اگر نخواهیم دو جدول در نظر نگیرد و هر دو را در یک جدول در نظر بگیرد بایستی چه کاری انجام دهیم ؟

در متد OnModelCreating بایستی کانفیگ زیر را اعمال کنیم .

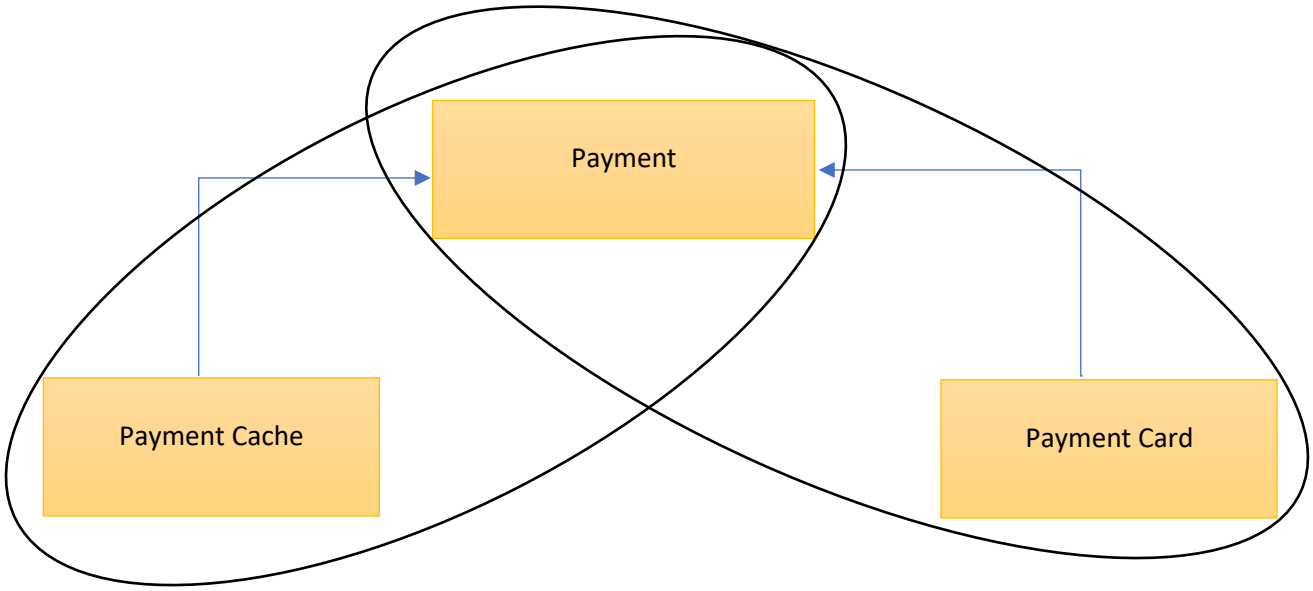
```
builder.Entity<PaymentCard()>
.HasBaseType(typeof(Payment));

builder.Entity<PaymentCash()>
.HasBaseType(typeof(Payment));
```

زمانی که دو کانفیگ بالا را اعمال می کنیم ، EF به صورت زیر عمل می کند.



در این صورت EF متوجه می شود که این دو کلاس فرزند یک وجه اشتراک دارند ، پس در این صورت دو جدول را یک جدول در نظر می گیرد.

اما به صورت پیش فرض و بدون کانفیگ خاصی به شکل زیر عمل می کند.



شکل بالا دو جدول در نظر گرفته می شود که پراپرتی های Payment در آن قرار دارند.

نکته : اگر ارث بری های من زیاد باشند ، اون موقع هم بایستی بیاییم و همه ی زیر کلاس ها را در OnModelCreating کانفیگ کنیم که BaseType دارند ؟

خب این موضوع یک مشکل هست . آیا EF برای موضوع تدبیری نکرده و اما اگر کرده بایستی برای رفع مشکل چه کاری را انجام دهیم ؟

برای حل این مشکل بایستی کانفیگ زیر را اعمال کنیم .

```
builder.Entity<Payment>()
.UseTPHMappingStrategy();
```

یک سناریویی وجود دارد که ما در Context فقط کلاس Payment را به عنوان Entity معرفی کرده ایم و نمی خواهیم به دو کلاس PaymentCash و PaymentCard به صورت مستقیم دسترسی داشته باشیم که بتوانیم بگوییم Context.PaymentCard ، اما اگر یادتون باشد ما در OnModelCreating ذکر کردیم که استراتژی TPH را برای Payment در نظر بگیر ، خب تکلیف مابقی چه می شود پس ؟

در اینجا ما بایستی به صورت غیر مستقیم دو انتیتی PaymentCash و PaymentCard را معرفی کنیم ، برای اینکار بایستی به شکل زیر عمل کنیم .

بایستی در متد OnModelCreating کد زیر را قرار دهیم .

```
builder.Entity<PaymentCard>();
builder.Entity<PaymentCash>();
```

حالا یک مشکل پیش میاد ، ما اگر بخواهیم از PaymentCard کوئری بگیرم چطور بایستی این کار را انجام دهیم ؟ چون ما که دیگر به صورت مستقیم به انتیتی PaymentCard دسترسی نداریم .
برای حل این موضوع بایستی به شکل زیر عمل کنیم .

```
context.Payment.OfType<PaymentCard>();
```

ما در TPH یک Shaddow Property داریم با نام Discriminator که نوع پیش فرض آن String می باشد و مقدار آن ، نام Type مربوطه می باشد .

EF امکانی را برای ما فراهم کرده تا بتوانیم نام و حتی تایپ این Shaddow Property را تغییر دهیم .
برای اینکار می توانیم به شکل زیر عمل کنیم .

```
builder.Entity<Payment>
.HasDiscriminator<PaymentType>(x => x.Type)
.HasValue<PaymentCard>(PaymentType.Card)
.HasValue<PaymentCash>(PaymentType.Cash)
.HasValue<Payment>(PaymentType.Unknown);
```

نکته : قبل از کانفیگ بالا، ابتدا بایستی یک پراپرتی به نام Type از نوع PaymentType بایستی تعریف کنیم.

در تنظیمات بالا برای Payment , PaymentCach , PaymentCard یک Discriminator از نوع Enum تعریف کردیم و برای هر کدام مقدار مربوطه اش را تنظیم کردیم ، اما زمانی هست که داریم به صورت DB First پیش میریم و هنوز نمی دانیم چه مقادیری داریم که بخواهیم مقدار مربوطه را در Enum اضافه کنیم .
در سناریو Payment ما فقط PaymentCash , PaymentCard و Unknown را داریم ، اما اگر یک مقدار در دیتابیس داشته باشیم که در این Enum موجود نباشد چه اتفاقی خواهد افتاد ؟
خب طبیعتا با خطا مواجه می شدیم ، اما خب آیا در اینجا EF به ما قابلیتی برای کنترل این موضوع داده یا خیر ؟

برای کنترل این موضوع در متد OnModelCreating بایستی به شکل زیر عمل کنیم تا با خطا مواجه نشویم .

```
builder.Entity<Payment>
.HasDiscriminator<PaymentType>(x => x.Type)
.HasValue<PaymentCard>(PaymentType.Card)
.HasValue<PaymentCash>(PaymentType.Cash)
.HasValue<Payment>(PaymentType.Unknown)
.IsComplete(false);
```

با قرار دادن IsComplete(false) در زمان واکشی دیتاها ، فقط دیتاهایی را واکشی می کند که برای آن مقدار مربوطه در Enum تعریف شده باشد ، با این کار دیگر دچار خطا نمی شویم .

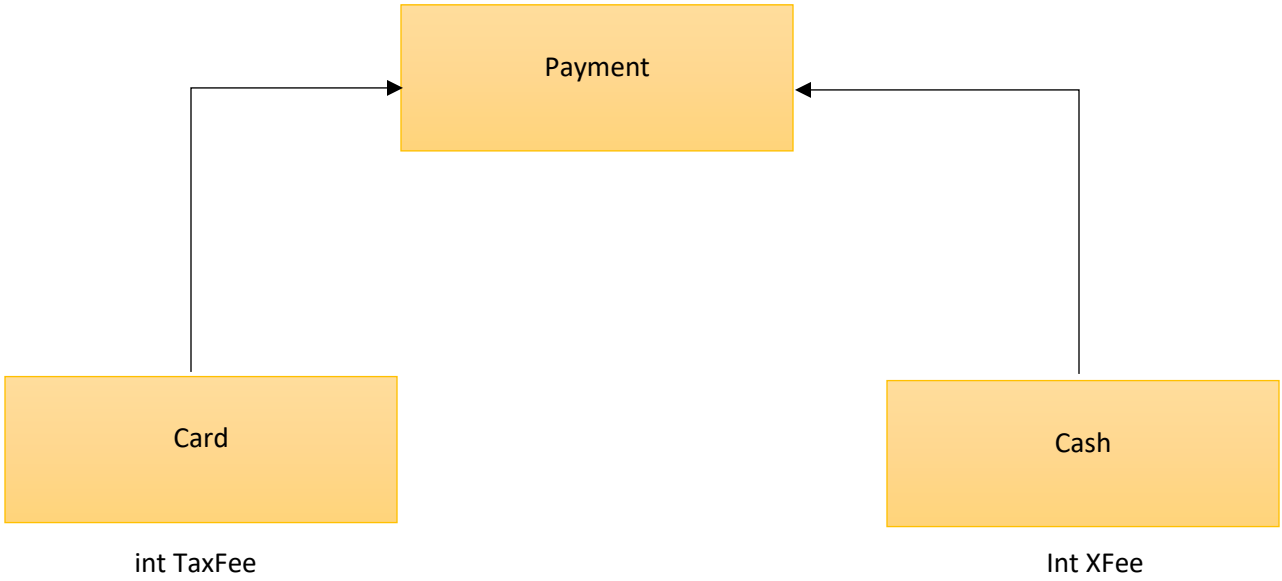
مشکلات TPH

1 - فهم TPH برای برنامه نویسان تازه کار یک خرده پیچیده است.

2 - در اینجا جدول PaymentCash به طور مثال ده تا پراپرتی دارد ، اما اگر بخواهیم به طور مثال با جدول PaymentCard کار کنیم و برای آن یک رکورد ثبت کنیم ، رکورد مربوطه ده تا فیلد Null دارد و عملا در این سناریو TPH به کارمان نمی آید.

نکته : TPH استراتژی پیش فرض EF برای بحث ارث بری می باشد.

بحث Shared Column در TPH



بحث اینجاست که ما دو تا پراپرتی در دو مدل مختلف داریم و حوزه کاریشون یکی و تایپ هر دو نیز یکی می باشد . آیا این امکان وجود دارد که برای این دو پراپرتی یک پراپرتی در نظر بگیریم ؟

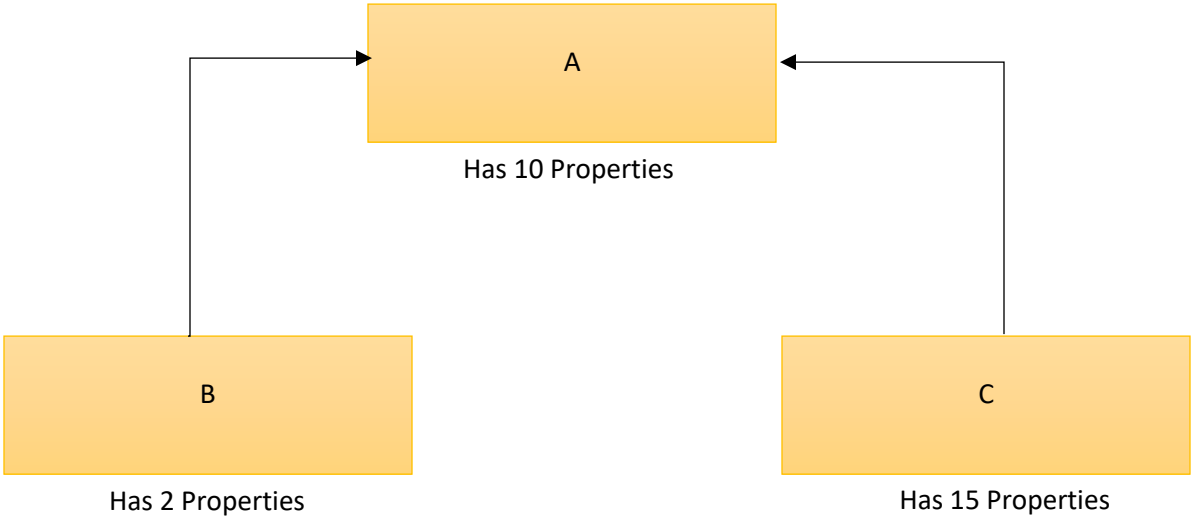
بله این امکان وجود دارد ، برای استفاده از این قابلیت بایستی به صورت زیر عمل کنیم .

```
builder.Entity<Card>()
.Property(x => x.TaxFee)
.HasColumnName("Fee");

builder.Entity<Cash>()
.Property(x => x.XFee)
.HasColumnName("Fee");
```

با اعمال کانفیگ بالا برای دو پراپرتی در دیتابیس یک فیلد در نظر می گیرد.

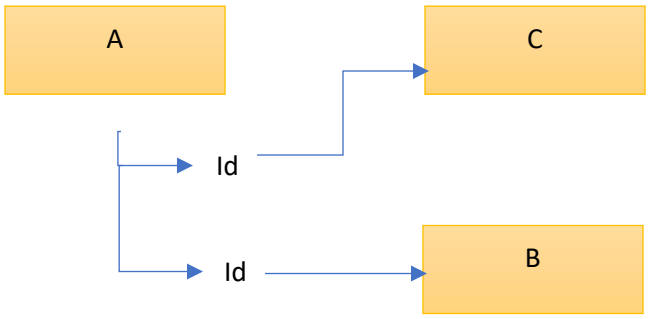
بحث TPT (Table Per Type)

داستان از این قرار است که ما یک مشکلی در استراتژی TPH داشتیم و مشکل این بود که اگر سناریوی زیر را داشته باشیم.

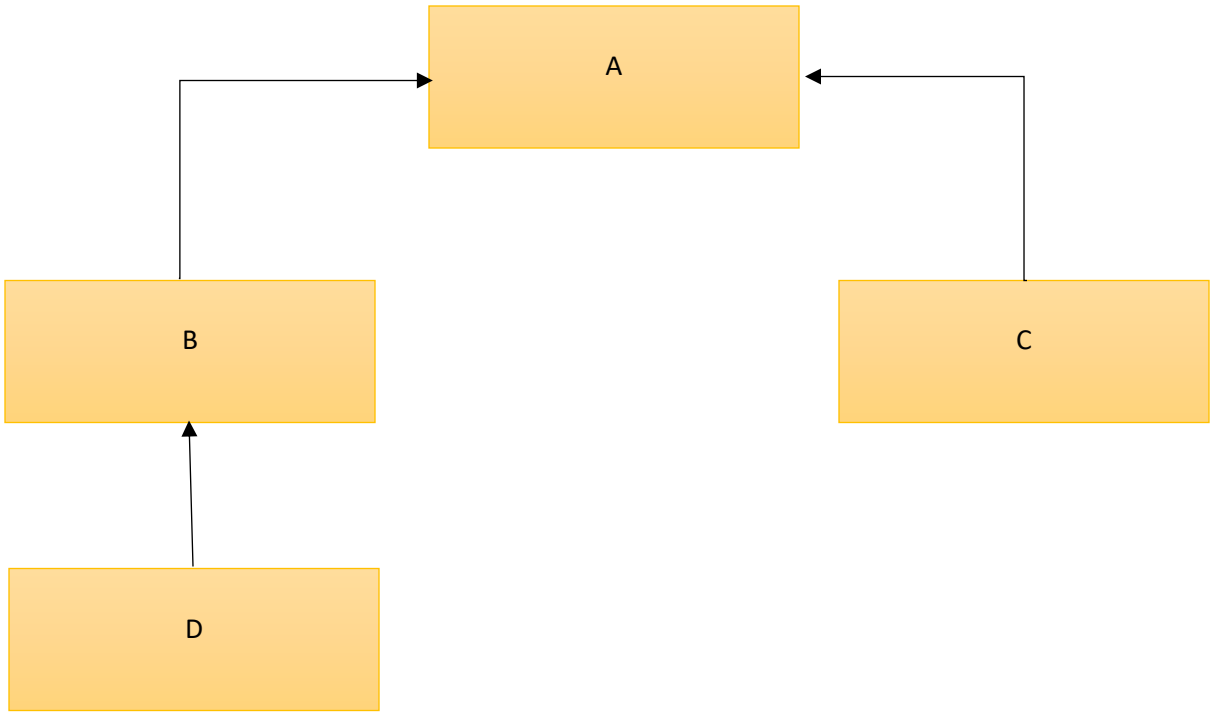


در این سناریو اگر بخواهیم روی جدول C عملیات Insert داشته باشیم ، به این دلیل که جدول B دارای 20 پراپرتی هست ، 20 تا فیلد Null پر می شود ، خب اینجا استراتژی TPT آمده تا این مشکل را رفع کند . اما به چه صورت ؟

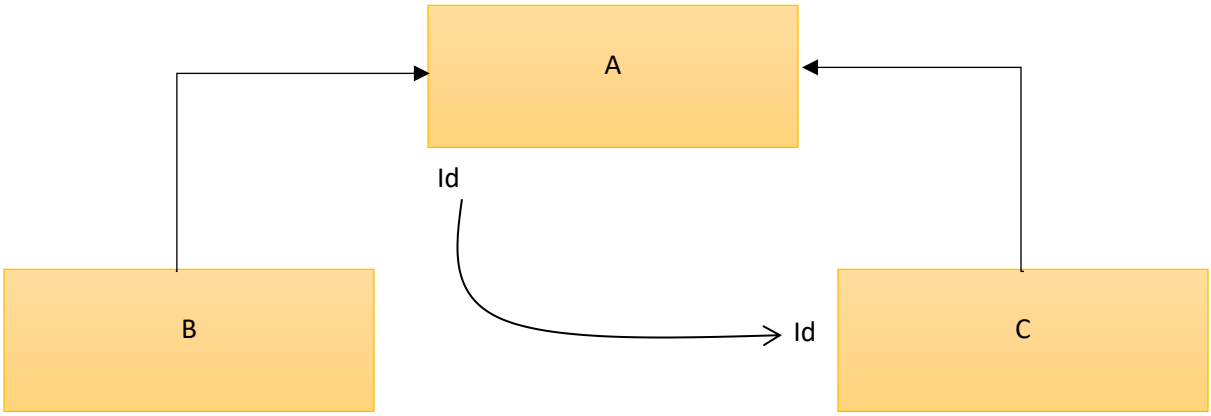
سه جدول A , B , C هر کدام یک جدول در نظر گرفته می شوند ، با این تفاوت که وقتی که می خواهیم به طور مثال در جدول C عملیات درج داشته باشیم ، ابتدا یک رکورد داخل جدول A ثبت می شود و Id سطر مربوطه را به عنوان مقدار Id در جدول C در نظر می گیرد و هم چنین مقادیر مربوط به جدول C را پر می کند.



نکته : یکی از مشکلاتی که استراتژی TPT دارد ، این هست که اگر سناریو زیر را داشته باشیم.



خب اگر بخواهیم دیتایی را دریافت کنیم ، EF برای دریافت دیتا از جداول Child مجبور هست که Join های متعددی را لحاظ کند و اگر این ارث بری ها زیاد شوند باعث افزایش Join می شود و این عمل باعث کاهش Performance می شود و ما رادچار مشکل خواهد کرد.



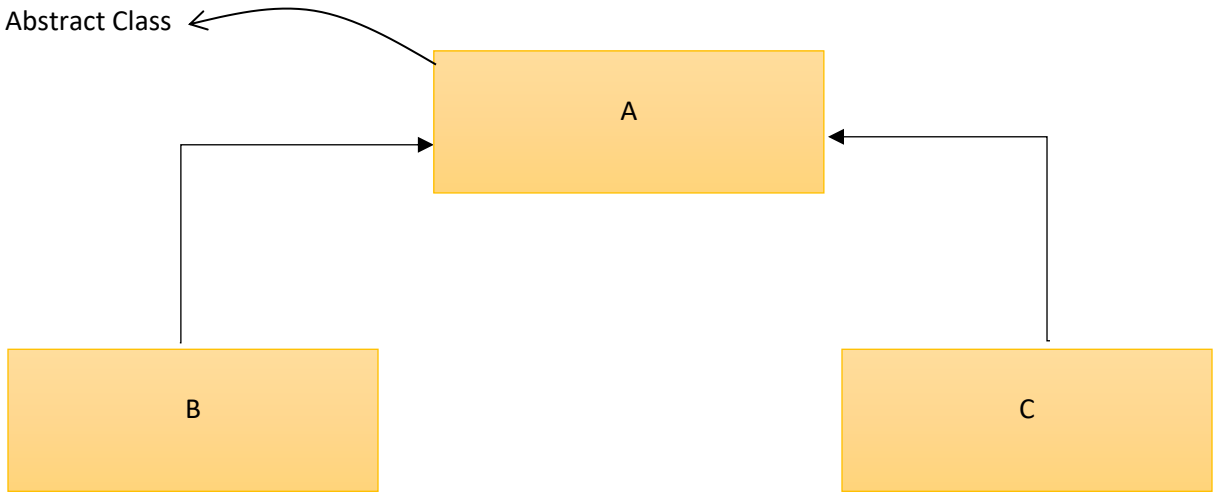
در مدل TPT ابتدا می آمد یک رکورد در جدول A درج می کرد ، سپس مقدار آن به عنوان Id در جدول مثلا C یا B لحاظ می شد .

خب اگر ما به صورت دستی به طور مثال در جدول C یک رکورد تستی وارد کنیم که مقدار Id آن در جدول A وجود نداشته باشد چه اتفاقی خواهد افتاد ؟

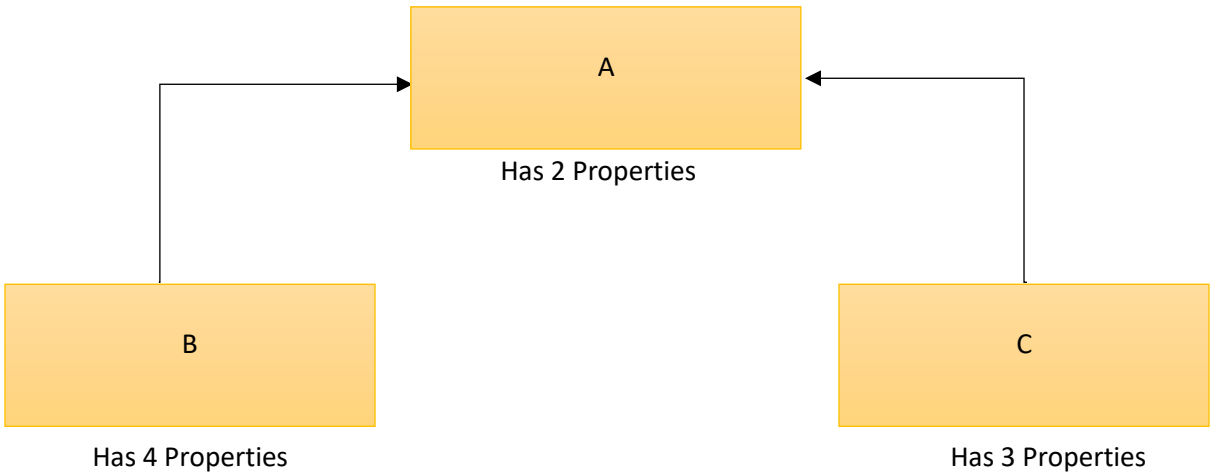
در اینجا در جدول C درست هست که فیلد Id به صورت Identity نیست ولی یک Contraint برای آن ست شده که این فیلد یک FK هست و به جدول A رفرنس دارد ، به همین دلیل ما نمی توانیم عددی در Id جدول C وارد نماییم که در جدول A وجود ندارد.

بحث TPC (Table Per Concreat)

نکته : در مدل زیر اگر کلاس A یک Abstract کلاس باشد تبدیل به جدول نمی شود.



در مدل TPC به ازای هر مدل یک جدول در نظر گرفته می شودو کلاس های Child پراپرتی های جدول پدر را هم به ارث می برند ، برای درک بهتر به مثال زیر توجه نمایید.



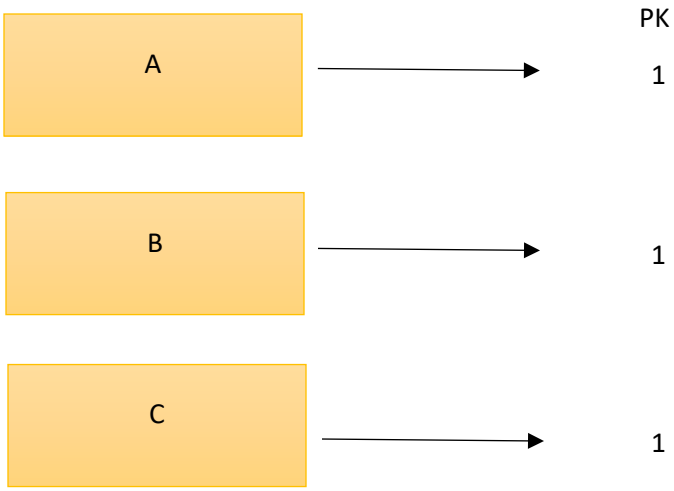
اگر کلاس A یک Abstract Class نباشد، به ازای آن یک جدول در نظر گرفته می شود که شامل دو پراپرتی می شود.

کلاس B شامل چهارتا پراپرتی می باشد که کلاس پدرش یعنی کلاس A هم دو پراپرتی ، پس یک جدول داریم به نام B که دارای شش فیلد می باشد و همین طور کلاس C که تبدیل به یک جدول می شود که دارای پنج فیلد می باشد.

نکته : مکانیزم پیش فرض EF در مواجه با تولید کلید اصلی در مدل TPC، Sequence می باشد ، چرا ؟

به توضیحات زیر توجه نمایید.

فرض کنید اگر کلید اصلی به صورت Identity بود چه اتفاقی می افتاد ؟!



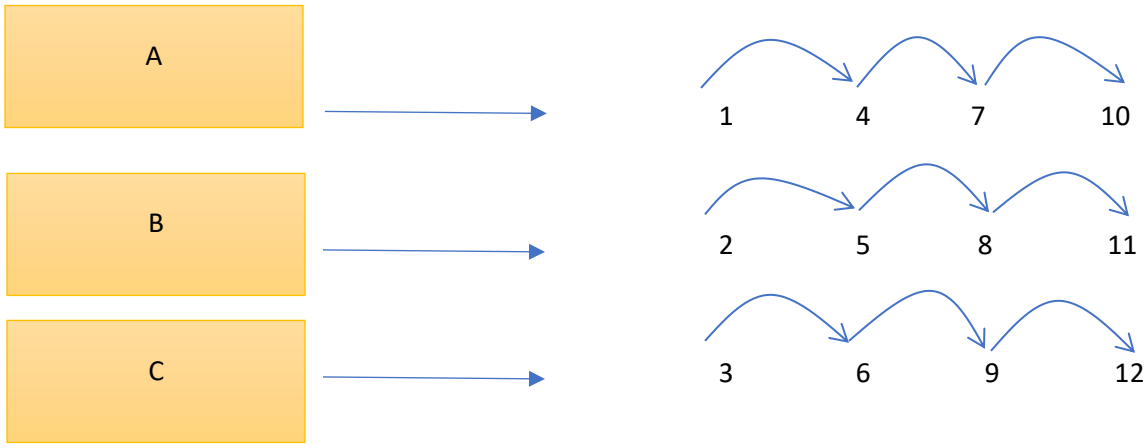
خب در اینجا اگر ما به طور مثال کلاس A یا B یا C را واکشی کنیم و بخواهیم کلاس B یا C را به کلاس A تبدیل کنیم ، چه اتفاقی خواهد افتاد ؟

سیستم Tracking از کجا متوجه می شود که این مقدار Id متعلق به کدامین جدول هست ؟

پس اگر کلاس A از دیتابیس واکشی شود سپس کلاس B یا C واکشی شود و تبدیل به کلاس A شود ، این اختلال عملا سیستم Tracking را دچار مشکل می کند.

نکته : خب ما مکانیزم پیش فرض EF برای تولید کلید اصلی در استراتژی TPC که Sequence بود را توضیح دادیم ، اما خب در برخی از دیتابیس ها این Sequence پشتیبانی نمی شود ، خب در این حالت بایستی چه کاری انجام داد ؟

در این حالت بایستی از سناریوی Identity استفاده کرد ولی با Seed و Incremental های مختلف که برای درک بهتر موضوع به شکل زیر توجه کنید.

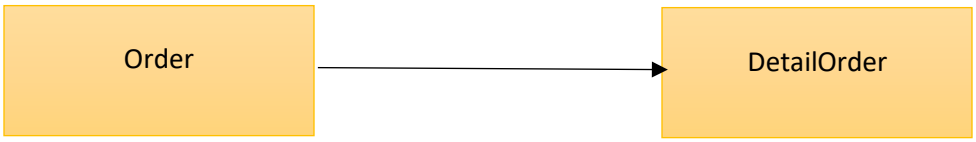


در اینجا می بینیم که هر کدام از جداول با مقادیر متفاوتی شروع شده اند و سه تا سه تا افزایش می یابند و با این سناریو دیگر تداخلی در کلیدهای اصلی کلاس با یکدیگر نخواهیم داشت.

مبحث Table Spliting یا Table Sharing

یکی از محدودیت هایی که ما در Owns داریم ، بحث کوئری زدن بر روی جداول Child هست که این امکان را نداریم .

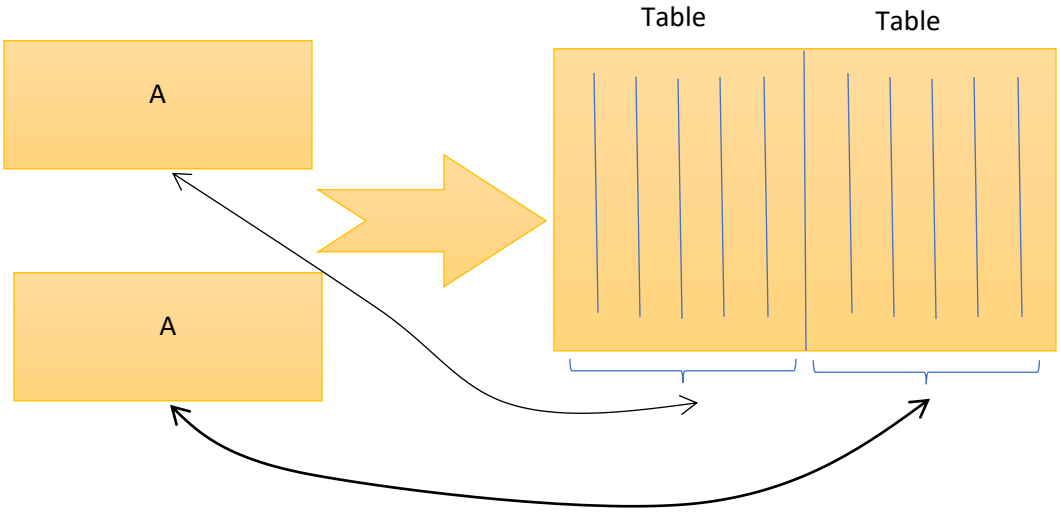
به مدل زیر توجه کنید.



برای مدل بالا می بایست کانفیگ کرد که Order یک Owns به نام OrderDetail دارد و مشکل از جایی شروع می شود که بخواهیم مستقیما روی OrderDetail کوئری بزنیم.

چون در Context فقط انتیتی Order معرفی شده ، پس ما نمی توانیم مستقیما روی OrderDetail کوئری بزنیم . برای حل این مشکل چه کاری می بایست انجام داد ؟

در این حالت Table Sharing می تواند به ما کمک کند.



اگر ما بر روی این جدول زیاد عملیات CUD داشته باشیم ، خب نیاز نیست دیگر پانزده تا فیلد جدول دومی را هم واکشی کنیم .

نکته : این مدل زمانی استفاده می شود که داریم به صورت DBFirst کار می کنیم.

در ادامه ی بحث Table Sharing در نهایت در برای کانفیگ کردن آن بایستی به شکل زیر عمل کنیم .

```
modelBuilder.Entity<OrderDetail>(c => )
{
    c.ToTable("Orders");
    c.Propeety(x => x.Status)
    .HasColumnName("Status");
 });

modelBuilder.Entity<Order>(c =>
{
   c.ToTable("Orders");

   c.Property(x => x.Status)
   .HasColumnName("Status");

   c.HasOne(x => x.OrderDetail)
   .WithOne()
   .HasForegnKey<OrderDetail>(x => x.Id);

   c.Navigation(x => x.OrderDetail)
    .IsRequired;()
});
```

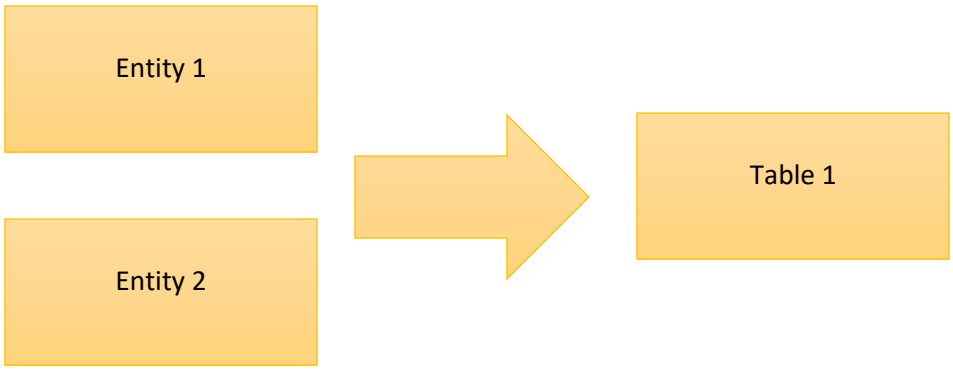
کانفیگ بالا برای این هست که بتوانیم در زمان کوئری گرفتن Include بزنیم و هم چنین EF بتواند به درستی دیتاهارا واکشی کند.

```
class Order {
  int Id
  OrderStatus Status
  OrderDetail OrderDetail
}

class OrderDetail {
  int Id
  OrderStatus OrderStatus
}
```

نکته کانفیگ بالا در جایی هست که ما Navigation را IsRequired در نظر گرفتیم که در ادامه توضیحات تکمیلی داده خواهد شد.

به صورت پیش فرض این Navigation به صورت Optional می باشد و چون در سمت Entity رابطه وجود دارد و چون ساخت OrderDetail هزینه بر هست ، پس EF یک سری Null Checking ها را لحاظ می کند که حتما پراپرتی های OrderDetail مقدار Null داشته باشند تا این آبجکت OrderDetail را نسازد ، اما اگر این Navigation به صورت Required باشد دیگر چک کردنی صورت نمی گیرد و کوئری به صورت عادی اجرا می گردد.

شماتیک Table Sharing

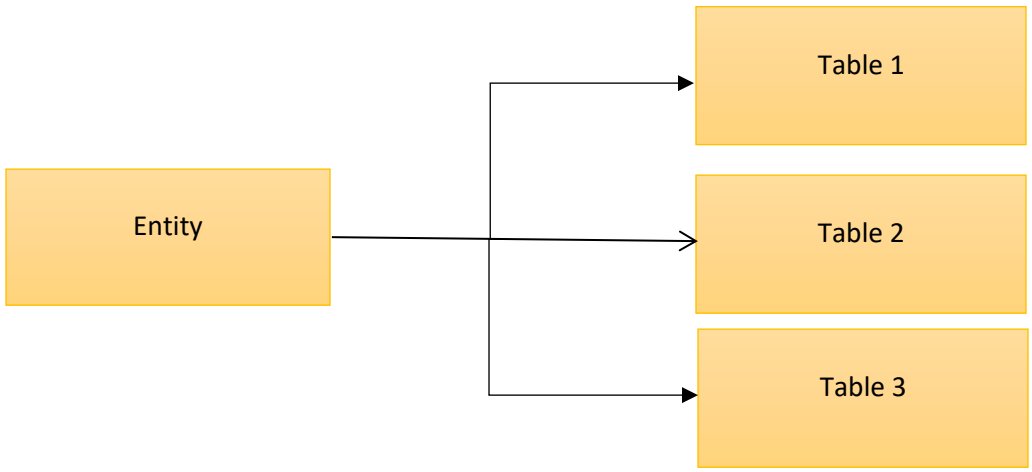


همانطور که مشاهده می کنید برای چندین Entity یک جدول در نظر گرفته شده است ، این سناریو شبیه TPH می باشد . اما دیگر ارث بری صورت نمی گیرد و می توان گفت در کل یک الگو برداری بوده است.

مبحث Entity Spliting

سناریو برعکس Table Spliting می باشد ، در این سناریو یک Entity به چندین جدول تبدیل می شود.

شماتیک Entity Spliting



بعد از ساخت Entity و معرفی آن به Context برای استفاده از قابلیت EntitySpliting در قسمت OnModelCreating بایستی کانفیگ زیر را اعمال کنیم .

```
builder.Entity<Customer>(c =>
{
  c.ToTable("Customers")
    .SplitToTable("PhoneNumbers",
     t =>  t.Property(x => x.Id)
           .HasColumnName("CustomerId");

    t.Property(x => x.PhoneNumber);
})
.SplitToTable("Address", t =>
{
    t.Property(x => x.Id)
     .HasColumnName("CustomerId");

    t.Property(x => x.Street);
    t.Property(x => x.City);
    t.Property(x => x.Country);
  });
});
```

مبحث Value Conversion

بعضی اوقات هست که ما در یک Entity یک پراپرتی از جنس Enum داریم ، خب می دانیم که این پراپرتی از جنس Enum در دیتابیس به صورت عددی ذخیره می شود و مشکل از جایی شروع می شود که مدام بایستی به یاد داشته باشیم که مثلا عدد صفر یا .... به چه معناست ؟!
در اینجا ما می توانیم به جای عدد مربوطه خود کلمه موجود در Enum را ذخیره کنیم و هنگام واکشی اطلاعات مجددا آن را به Enum مربوطه تبدیل کنیم.
برای انجام این کار می توانیم به شکل زیر عمل کنیم.

```
builder.Entity<Order>
.Property(x => x.State)
.HasConversion(
clrToDb =>  clrToDb.ToString(),
dbToClr => (OrderState)Enum.Parse(typeof(OrderState), dbToClr);
);
```

ما می تونیم روش قبل را به صورت زیر که ابتدا برای تعریف Value Converter یک کلاس ساخته و آن را به EF معرفی می کنیم .

```
public class OrderStateConverter : ValueConverter<OrderState, string>
{
   public OrderStateConverter()
     : base (
       clrToDb =>  clrToDb.ToString(),
       dbToClr => (OrderState)Enum.Parse(typeof(OrderState), dbToClr))
      {}
}
```

نحوه استفاده از کلاس بالا :

```
builder.Entity<Order>
.Property(x => x.State)
.HasConversion<OrderStateConverter>();
```

خود EF به صورت تو کار یک سری Predefine Conversion ها دارد و در اکثر موارد نیازی نیست خودمان دست به کار شویم و خودمان آن را پیاده سازی کنیم.
به طور مثال در مثال قبلی برای OrderState نیازی نیست خودمان برای آن Converter بنویسیم ، چون که خود EF آن را دارد که به شکل زیر می توانیم از آن استفاده کنیم .

```
builder.Entity<OrderState>
.Property(x => x.State)
.HasConversion<string>();
```

خب ما تا اینجا متوجه شدیم که Converter ها چه کارهایی را می توانند انجام دهند ، ولی یک مشکلی وجود دارد ، اگر من مثلا یک ComplexType در چندین Entity دارم مثلا Money Type مدام بایستی کانفیگ کنیم که در فلان Entity فلان Property دارای این Conversion می باشد ، خب ما می تونیم این Converter خودمان را به شکل گلوبال تعریف کنیم ، به چه شکل ؟
ابتدا بایستی ConfigureConventions را در Context پروژه Override کنیم و سپس به شکل زیر بایستی عمل کنیم :

```
builder.Properties<OrderState>()
.HasConvension<OrdetStateConverter>();
```

نکته : یک محدودیتی در گلوبال Convension ها داریم این هست که نمی توانیم بر اساس یک سری شرایط یک Converter را اعمال کنیم ، به طور مثال اگر Price > 0 بود بیاد و کانفیگ را اعمال کند.

مبحث Shadow Property

پراپرتی هایی هستند که در سمت کد پروژه موجود نمی باشند اما در دیتابیس موجود می باشند و اگر بخواهیم یک Shadow Property تعریف کنیم و از آن استفاده کنیم ، بایستی به شکل زیر عمل کنیم .

بایستی در متد OnModelCreating موجود در Context پروژه کانفیگ زیر را اعمال کنیم.

```
builder.Entity<Order>()
.Property<int>("OrderCode");
```

➔ How to use ?

```
var orders =
  context.Orders
  .where(x => EF.Property<int>(x, "OrderCode") == 15)
.ToList();
```

نکته : از آنجایی که Shadow Property ها را در سمت کد نداریم اما در سمت دیتابیس داریم ، متوجه می شویم که مدیریت و ساخت و بیلد آن را خود WF مدیریت می کند.

بریم سر وقت Backing Field و اینکه چه تفاوتی با Shadow Property ها دارند.

Backing Field

این نوع از پراپرتی شبیه به Shadow Property ها هستند ، اما با یک سری تفاوت ها که در ادامه آن ها را مطرح می کنیم.

پراپرتی هایی که Shaow بودند را خود EF مدیریت می کرد (منظور از مدیریت منظورمان مدیریت کردن get و set هست) اما مدیریت Backing Field ها دیگر به عهده خودمان می باشد .

```
public class Order
{
    private int orderCode;
    public int CalculatedOrderCode   { get => orderCode; }
}
```

در قسمت بالا کلاس مدنظرمون را تعریف کردیم و برای ادامه می بایست آن را در متد OnModelCreating موجود در Context پروژه کانفیگ کرد.

```
builder.Entity<Order>()
.Property(x => x.CalculatedOrderCode)
.HasField("orderCode")
.HasColumnName("OrderCode");
```

نکته : BackingField در واقع از همان ویژگی Backing Field در زبان سی شارپ استفاده می کند و به همین خاطر است که اسم آن در EF هم Backing Field نام گذاری شده است.

در اینجا یک سوال مطرح می شود !
در مثال Backing Field ما یک فیلد با نوع int و سطح دسترسی private و یک پراپرتی public که فقط get دارد ، خب موقعی که می خواهیم دیتا از دیتابیس واکشی کنیم این فیلد private به چه صورت مقدار دهی می شود ؟
خب در جواب به این سوال ابتدا بایستی یک تعریف کلی از سطح دسترسی private داشته باشیم که این نوع سطح دسترسی به این گونه هست که ما فقط می توانیم از داخل کلاس به متد یا پراپرتی تعریف شده دسترسی داشته باشیم و از بیرون کلاس قابل دسترس نیست .
اما خب این نکته در مورد سطح دسترسی private فقط در زمان Comoile صدق می کند و نه در زمان RunTime.

چه قابلیتی می تواند این عمل را در زمان RunTime انجام دهد ؟
در زبان سی شارپ Reflection می تواند این کار را برای ما انجام دهد.

نکاتی در بحث **Backing Field**

هنگاهی که در متد OnModelCreating در حال تعریف یک BackingField هستیم می توانیم یک سری Access Mode برای آن تعریف کنیم .

```
builder.Entity<Order>()
.Property(x => x.CaculateOrderCode)
.HasField("OrderCode")
.HasColumnName("OrderCode")
.UsePropertyAccessMode(PropertyAccessMode.Field);
```

در مثال بالا از یک Enum با نام PropertyAccessMode استفاده کردیم که در ادامه می خواهیم هر یک از Member های آن را توضیح دهیم.

1 - Property

مقدار را روی پراپرتی مربوطه set می کند ، اما اگر پراپرتی مربوطه set نداشته باشد و فقط get داشته باشد exception رخ می دهد.

2 - PreferProperty

مانند Property می باشد ، اما اگر پراپرتی مربوطه set نداشته باشد دیگر exception رخ نمی دهد و سعی می کند با Reflection فیلد مربوطه را مقدار دهی کند.

3 - Field

به صورت پیش فرض روی این حالت می باشد و فیلد مربوطه را مقدار دهی می کند.

4 - PreferField

به این صورت می باشد که اگر روی این حالت باشد ، ترجیح ما روی Field هستش ، در صورتی که به صورت پیش فرض فیلد مربوطه را مقدار دهی می کند و اگر فیلد مربوطه را نتوانست پیدا کند ، پراپرتی مربوطه را مقدار دهی می کند.

بحث AsSplitQuery

کوئری زیر را در نظر بگیرید :

```
var orders =
 context.orders
.Include(x => x.OrderItem)
.ThenInclude(x =>    x.OrderDimention);
```

در کوئری بالا سه انتیتی درگیر است و در اجرای کوئری ضربدر دکارتی اتفاق می افتد که اگر دیتاهای درج شده زیاد باشند کوئری به کندی اجرا خواهد شد ، به همین دلیل در این سناریو از AsSplitQuery می توانیم استفاده کنیم .
به طور مثال در کوئری بالا سه انتیتی درگیر هستند و تمامی ننایج با یک درخواست برگردانده نمی شوند و برای دریافت تمامی نتایج با سه درخواست به دیتابیس دیتاهای هر جدول را به طور کامل دریافت می کند.

نکته : زمانی که ضرب درکارتی اتفاق می افتد Dupplication رخ می دهد و رکوردهای تکراری ایجاد می شوند در صورتی که در AsSplitQuery این اتفاق رخ نمی دهد.

نکته : اگر شبکه ما و دیتابیس ما سرعت پایینی دارد سناریوی AsSplitQuery می تواند مشکل پرفرمنسی به وجود بیاورد.

نکته : ما می توانیم کانفیگ کنیم که EF کوئری ها را به صورت Single Query و یا Split Query ایجاد کند ، که به صورت پیش فرض کوئری ها به صورت Single و در قالب یک کوئری ایجاد می شوند.

تفاوت **FirstOrDefault** و **Find**

در یک سری از موارد ما نیاز داریم که حتما دیتاها از دیتابیس واکشی شوند در این سناریوها از FirstOrDefault استفاده می کنیم و اما اگر بخواهیم از First Level Cache استفاده کنیم و خود EF ابتدا Memory را چک کرده و در صورت نبودن در Memory آن را از دیتابیس واکشی کند ، که در این سناریوها می توانیم از Find استفاده کنیم .
در مجموع Find از سیستم Tracking استفاده می کند ، اما FirstOrDefault از سیستم Tracking استفاده نمی کند و همیشه دیتاها را از دیتابیس واکشی می کند .

بررسی خروجی دو متد **FindAsync** و **FirstOrDefaultAsync**

۱ - **FirstOrDefaultAsync**

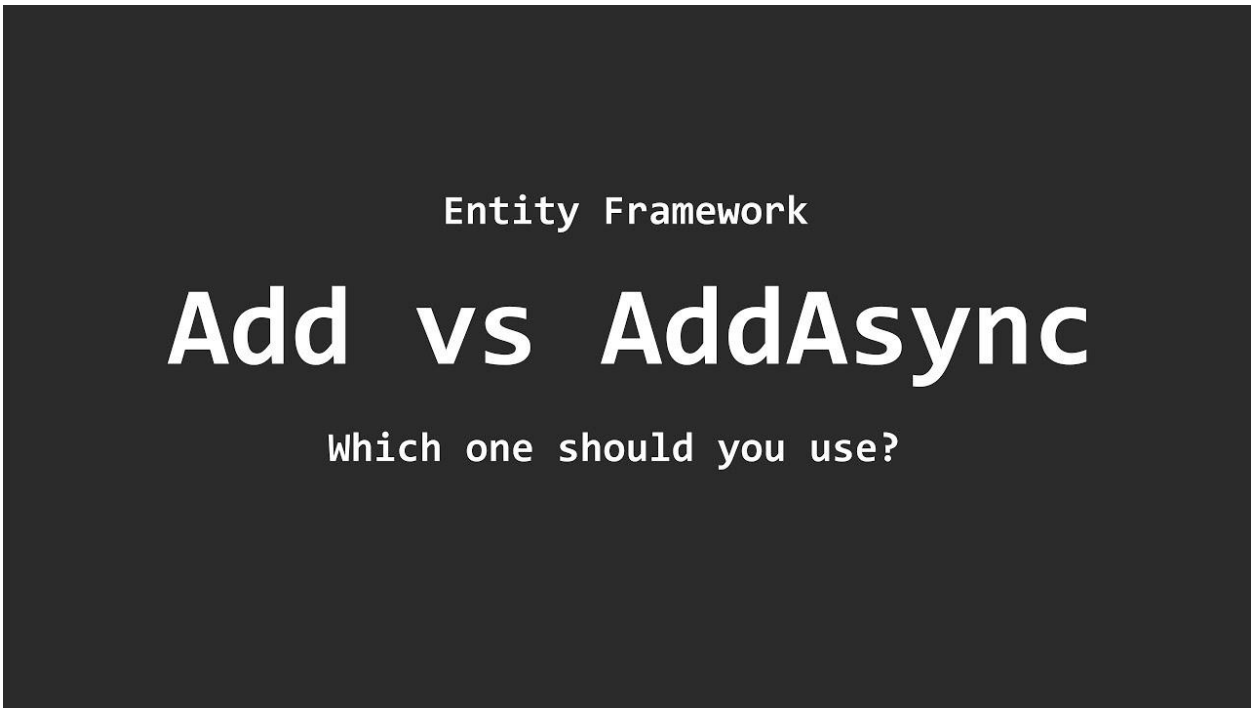
این متد به دلیل اینکه از سیستم Tracking استفاده نمی کند خروجی آن از نوع Task<T> می باشد.

۲ - **FindAsync**

این متد به دلیل اینکه از سیستم Tracking استفاده می کند خروجی آن از نوع ValueTask<T> می باشد.

بحث Add و AddAsync

چه زمانی باید از Add و چه زمانی باید از AddAsync استفاده کرد ؟



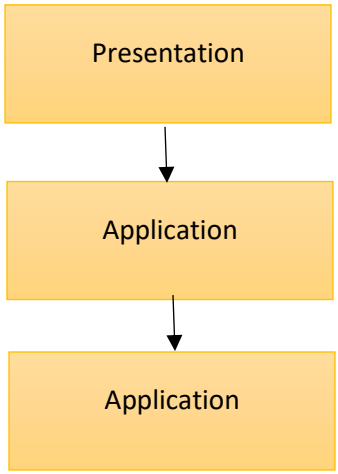
زمانی که یک Add ساده داریم و هیچ بیزنسی که بخواهیم از دیتابیس مقداری را واکشی کنیم نداریم و فقط تنها یک Add ساده هست بایستی از Add استفاده کنیم.

زمانی باید از AddAsync استفاده کرد که یک Value Generation داریم ، به طور مثال می خواهیم Id که یک فیلد PK هست را از دیتابیس واکشی کرده و یا هر مقداری که در زمان Add بایستی از دیتابیس واکشی شود.

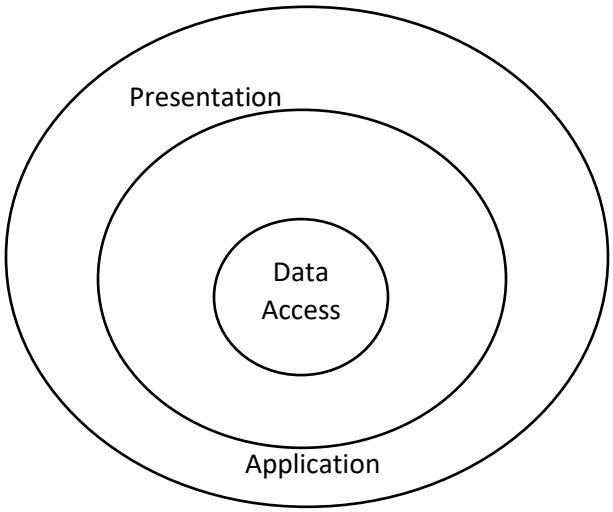
به طور مثال اگر بخواهیم یک مثال واقعی بزنیم می توان به الگوریتم Hilo اشاره کرد که الگوریتی برای تولید کلید اصلی هست و بایستی از AddAsync استفاده کرد.

بحث **Unit Of Work** و الگوی **Repository**

خب در ابتدای کار ما در لایه Application می خواهیم کدی بنویسیم که نیاز به DbContext دارد ، به همین خاطر در لایه Application نیاز داریم تا لایه Infrastructure را رفرنس دهیم اما در اینجا یک مشکلی پیش میاد ، چه مشکلی ؟
زمانی که Infrastructure را به لایه Application رفرنس دهیم ، وابستگی لایه ها به شکل زیر می شود.



ساختار بالا ساختار N Layer و Data Centric می باشد.
خب ما از معماری Clean استفاده می کنیم ، اما زمانی که Infrastructure را به Application رفرنس می دهیم ، در واقع داریم از معماری N Layer پیروی می کنیم.

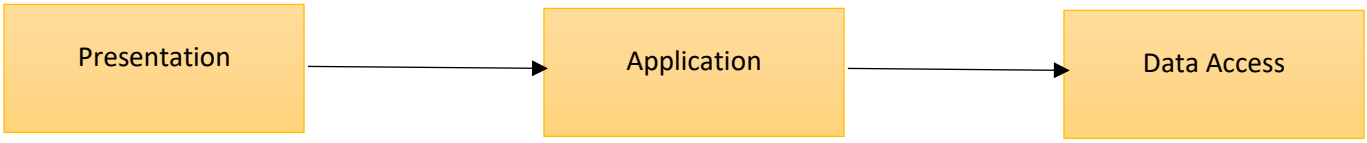


Data Centric

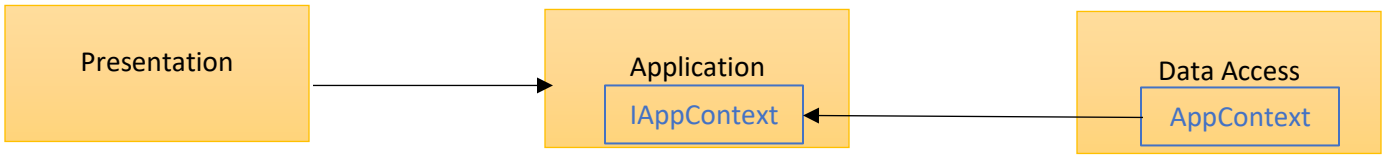
در اینجا ما متوجه شدیم که رفرنس دادن لایه Infrastructure به لایه Application اشتباه می باشد ، خب سوال پیش میاد که چطور میشه این مشکل را حل کرد ؟

در ابتدا بایستی در لایه Application رفرنس لایه Infrastructure را حذف کرد.

برای رفع این مشکل بایستی به معماری Hexagonal رجوع کنیم.



در معماری Hexagonal به این موضوع اشاره کرده که برای رفع این مشکل بایستی وابستگی لایه Application به Data Access را بایستی Reverse کرد ، برای درک بهتر به شکل زیر توجه کنید.



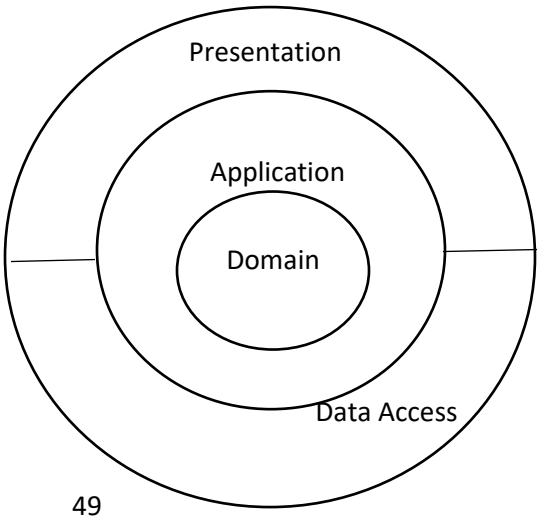
در معماری Hexagonal توضیح داده که برای رفع مشکل بایستی در لایه Application یک اینترفیس ساخت و پیاده سازی آن در لایه Data Access انجام شود ، با این کار ما وابستگی را معکوس کردیم.

خب ما در لایه Application اینترفیس IAppContext را تعریف کردیم ، ابتدا بایستی متد SaveChanges را داشته باشد و دو اینکه به طور مثال ما داریم بر روی User کار می کنیم ، بایستی شامل DbSet<User> نیز باشد.

نکته : در لایه Application بایستی از همین اینترفیس به عنوان Context استفاده کرد و می توان آن را در صورت نیاز اینجکت کرد.

نکته : برای پیاده سازی این اینترفیس بایستی لایه در Infrastructure لایه ی Application را رفرنس داد.

اگر بخواهیم به صورت شماتیک معماری پروژه را رسم کنیم ، به شکل زیر می رسیم.



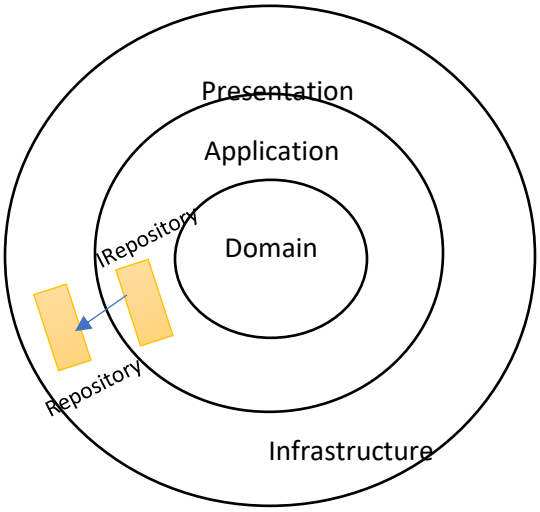
با توجه به کارهایی که انجام شد ، معماری ما مبتنی بر Clean شد ، چرا ؟

چون در لایه Application تنها Use Case هندل می شود.

در لایه Application به طور مثال اگر بخواهیم برای User یک سری Command و یا Query های مختلف بنویسیم ، نیاز داریم تا یک پراپرتی از جنس <DbSet<User را در اینترفیس IAppContext داشته باشیم تا بتوانیم روی این انتیتی یک سری کارها انجام دهیم اما خب در اینجا با یک مشکل جدیدی مواجه می شویم ، چه مشکلی ؟

خب ما در اینترفیس IAppContext موجود در لایه Application آمدیم و از <DbSet<User استفاده کردیم و این DbSet متعلق به پکیج EfFCore هست و ما لایه Application را به یک پکیج که در لایه Infrastructure استفاده شده ، وابسته کردیم و در واقع Coupling به وجود آمده است و اگر بخواهیم در آینده ORM پروژه را تغییر دهیم ، ORM مربوطه اصلا نمی داند DbSet چه چیزی هست و این اتفاق باعث می شود تا جاهای مختلف پروژه دست خوش تغییرات شود و این اصلا اصولی نیست و بر خلاف قوانین معماری Clean می باشد.

خب تا اینجای کار ما درگیر یک وابستگی شده ایم و آن هم نصب بودن پکیج EFCore در لایه Application می باشد و برای رفع این موضوع بایستی از الگوی Repoaitory استفاده کرد.



از آنجایی که لایه Application نبایستی درگیر نحوه پیاده سازی و تکنولوژی استفاده شده در لایه Infrastructure باشد ، بایستی از یک اینترفیس استفاده کند و نحوه پیاده سازی آن را به لایه Infrastructure بسپارد.

نکات :

در لایه Application تنها مشکل فقط وجود DbSet نیست ، بلکه مشکلات دیگری نیز وجود دارد که در ادامه به آن ها اشاره خواهد شد.

1 - مشکل Functionality

خب ما در لایه Application یک Handler برای انتیتی User نوشتیم که وظیفه Add را داشته باشد ، اما زمانی که context.User را تایپ می کنیم ، می توانیم Remove هم کنیم و در صورتی که ما نمی خواهیم انتیتی User قابلیت Remove داشته باشد ، چرا این اتفاق می افتد ؟ چون User یک DbSet هست.

2 - مشکل Testability

اگر بخواهیم برای Handler موجود که وظیفه ثبت کاربر را دارد تست بنویسیم ، این امکان را نداریم ، چون نمی توانیم DbSet را Mock کنیم.

نکته : بهترین محل برای تعریف Repository و تعریف Scope هر انتیتی لایه Domain می باشد ، پس در لایه Domain یک پوشه با نام Repoaitories ساخته و اینترفیس Repoaitory هر انتیتی را در آن تعریف می کنیم و در انتها پیاده سازی آن ها در لایه Infrastructure صورت می گیرد .

تا اینجا ما یک قدم به سمت معماری Clean نزدیک تر شدیم ، چون دیگر در لایه Application نه خبری از DbSet هست و نه دیگر دسترسی مستقیم به AppContext دارد.

آیا تمام مشکلات حل شد و هیچ گونه مشکل دیگری وجود ندارد ؟!

بریم در ادامه ببینیم مشکلی وجود دارد یا خیر !؟

تا اینجای کار ما آمدیم Reposotpry های مورد نیاز را پیاده سازی کردیم و در Handler های موجود در لایه Applocation استفاده کردیم و از دسترسی مستقیم به Context جلوگیری کردیم ، در نهایت Handler ما به شکل زیر می باشد.

```
userReposotory.Add(new User {});
userReposotory.SaveChanges();
```

در اینجا یک سوالی پیش می آید !؟
ما ابتدا User را Add و سپس متد SaveChanges را اجرا کردیم ، آیا ما این تضمین را می توانیم بدهیم که در زمان فراخوانی این متد EF تنها فقط کوئری Insert کردن User را بسازد و تغییرات دیگری در دیتابیس صورت نگیرد ؟!
خب در جواب بایستی باید بگوییم که ما این تضمین را نمی توانیم بدهیم ، چرا ؟
چون در پشت صحنه ChangeTracker وجود دارد و در زمان فراخوانی متد SaveChanges در پشت صحنه متد DetectChanges فراخوانی می شود ، پس در این صورت تمامی تغییرات در دیتابیس اعمال می شود.
تا اینجای کار ما با مشکل فراخوانی متد SaveChanges رو به رو شدیم و بایستی یک راه حل برای آن در نظر گرفت که در ادامه به آن می پردازیم .

مشکل دیگری که در استفاده از الگوی Repository وجود دارد این هست که ما مشکل Duplicate Code داریم ! مثلا اگر سی تا Repository داریم و در تمامی آن ها متد Add یا Delete داریم ، بایستی این دو متد را در تمامی آن ها پیاده سازی کنیم که این یک کار تکراری هست و برای حل مشکل تکرار کد ناخودآگاه به سمت الگوی Generic Repository می رویم.

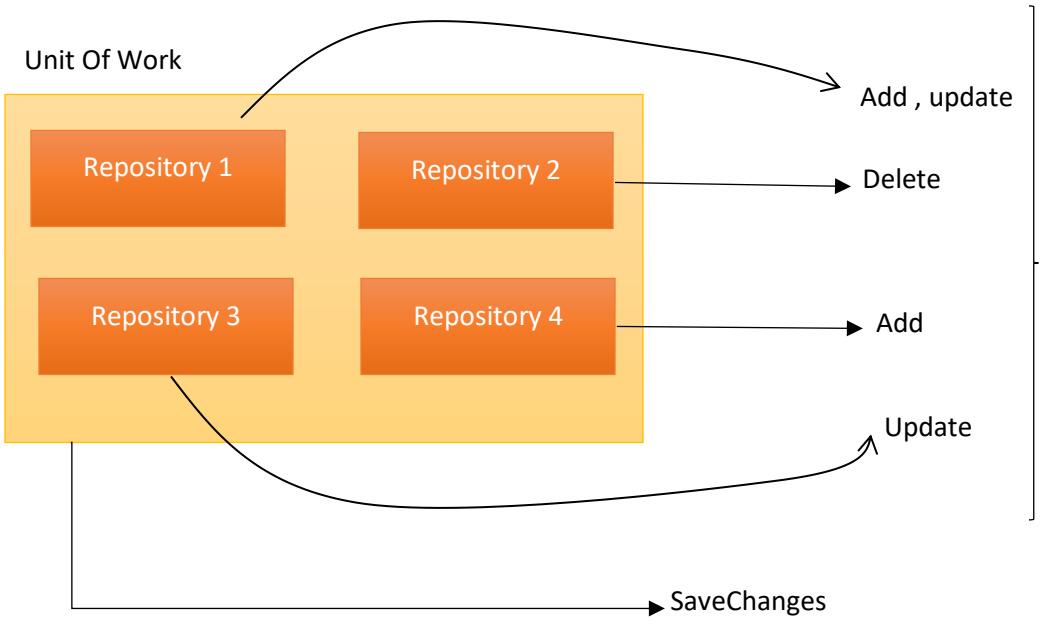
نکته : همان طور که در قبل هم توضیح داده شد ، وجود متد SaveChanges در داخل خود هر Repository اشتباه می باشد .
نکته : ما در الگوی Repository مشکل Duplicate Code داشتیم که برای حل آن به سمت Generic Repoaitory رفتیم .

خب حالا محل قرارگیری اینترفیس Generic Repository در چه لایه ای از پروژه هست ؟
از آنجایی که Generic Repoaitory برای تمامی Entity مورد استفاده می باشد ، می توان اینترفیس مربوطه را در لایه Application قرار داد .

Application Layer => Common => Interfaces => IGenericRepository

و در ادامه می توان پیاده سازی این اینترفیس را در لایه Infrastructure قرار داد.

یکی از مشکلاتی که داشتیم وجود متد SaveChanges درون خود Repoaitory می باشد ، که سوال پیش میاد محل پیاده سازی صحیح این متد کجا هست ؟!
در جواب به این سوال بایستی به الگوی Unit Of Work رجوع کنیم ، خب بریم ببینیم که به چه صورت هست.



نکته: داخل Unit Of Work تمامی Repoaitory های ما به صورت ReadOnly و متد SaveChanges وجود دارند.

```
interface IUnitOfWork
{
   IGenericRepository<User> { get; }
   Task<int> SaveChangesAsync();
}

class UnitOfWork (AppContext dbContext) : IUnitOfWork
{
   public IGenericRepository<User>  User => new   GenericRepository(dbContext);
   public Task<int> SaveChangesAsync() => dbContext.SaveChangesAsync();
}
```

تا به اینجای کار ما Generic Repoaitory و الگوی UnitOfWork را پیاده سازی کردیم .

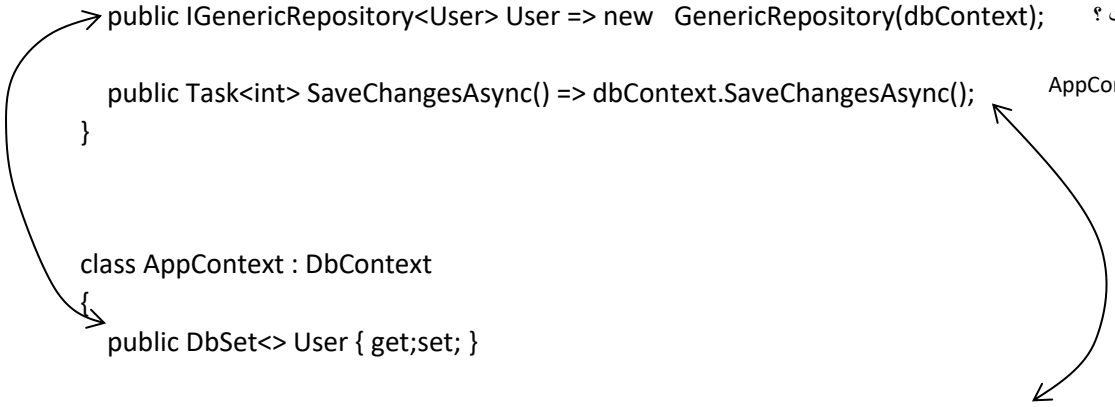
آیا مشکلات ما حل شدند یا خیر ، یعنی هم چنان مشکلاتی وجود دارند ؟ بریم بررسی کنیم که آیا مشکلی وجود دارد یا خیر.

برای درک بهتر به بررسی زیر دقت فرمایید.

```
class UnitOfWork(AppContext dbContext) : IUnitOfWork
{
   public IGenericRepository<User> User => new   GenericRepository(dbContext);

   public Task<int> SaveChangesAsync() => dbContext.SaveChangesAsync();
}
```

آیا این پراپرتی همان DbSet نیست ؟

آیا این متد همان متد موجود در AppContext نیست ؟!



```
class AppContext : DbContext
{
   public DbSet<> User { get;set; }

   public override Task<int> SaveChangesAsync() => base.SaveChangesAsync();
}
```

در نهایت ما اینجا به یک Anti Pattern رسیدیم و در واقع پیاده سازی یک Abstraction بر روی یک Abstraction.

در AppContext اگر دقت باشید ترکیبی از الگوی Repository و Unit Of Work می باشد.

بریم با هم ببینیم که این دو الگو کجاهای این DbContext هستند.

```
public class AppContext : DbContext
{
  public DbSet<User> User   {get;set;}

  public override Task SaveChangesAsync() => base.SaveChangesAsync;()
}
```

اگر به ساختار AppContext دقت کنید به همان الگوی Unit Of Work می رسیم ، چطور ؟

خب در اینجا DbSet ها همان Repository های ما هستند که به ما قابلیت های Add,Update,Remove و ... را می دهند و در انتها متد SaveChangesAsync موجود هست که دقیقا همان ساختار Unit Of Work را دارد.

نکته : آیا ما اصلا نبایستی از Generic Repository استفاده کنیم ؟

در جواب بایستی گفت که در برخی از سیستم ها که Crud Base هستند و پیچیدگی زیادی ندارند و هم چنین قرار نیست پروژه خیلی بزرگی باشند می توان از آن ها استفاده کرد.

ما یک Handler نوشتیم برای Add کردن User و در آن IUnitOfWork را اینجکت کردیم . خب ما در این Handler تنها می خواهیم یک User را Add کنیم و نیازی به OrderRepository و .... نداریم ، چرا بایستی در اینجا یک آبجکت بزرگ را اینجکت کنیم ؟!

نکته : ما در قبل مشکل دسترسی مستقیم به DbSet داشتیم که هر کاری که می خواستیم می توانستیم انجام دهیم ، در صورتی که به طور مثال User ما قابلیت Remove ندارد اما دیدیم که می توانیم آن را Remove کنیم و دیگر Scope آن دامنه را غیر قابل کنترل می کرد و نکته اینجاست زمانی که از یک Anti Pattern مثل Generic Repoaitory استفاده می کنیم مجددا با این مشکل مواجه می شویم که Scope دامنه غیر قابل کنترل می شود .

نکته : در Generic Repoaitory ما بایستی کلاس هایی را معرفی کنیم که در Context به صورت DbSet معرفی شده اند وگرنه در زمان RunTime با خطا مواجه می شویم ، خب برای حل این مشکل بایستی چه کاری انجام داد ؟

خب ما در اینجا میتوانیم یک اینترفیس با نام IEntity تعریف کنیم و در Entity از این اینترفیس ارث بری کنیم و در Generic Repository ذکر کنیم که TEntityحتما بایستی از IEntity ارث بری کرده باشد.

اما در اینجا یک نکته وجود دارد که در مستندات مایکروسافت نیز به آن اشاره شده است ، نکته ای در مورد اینترفیس هایی که هیچ عضوی در آن تعریف نشده و اصطلاحا به آن ها Empty Interface گفته می شود.

اما برویم سراغ نکته ی AVoid Empty Interface که در بالا به آن اشاره نمودیم.

به معنی اینکه بایستی پرهیز کنیم از تعریف اینترفیس هایی که هیچ ممبری ندارند ، چون ذات اینترفیس آمده تا یک رفتار و یا یک Contract را به وجود بیاورد.

اما یک نکته و استثنایی برای تعریف Empty Interface ها برای متمایز کردن یک یا چندین کلاس از مابقی کلاس ها وجود دارد که اگر این شناسایی در زمان RunTime باشد بایستی از Custome Attribute استفاده شود وگرنه در زمان Compile استفاده از اینترفیس ها قابل قبول می باشند.

خب در آخر بحث Repository به کجا رسید ؟ نتیجه چه شد و بایستی چه کرد تا پیاده سازی درستی داشت ؟

بریم ببینیم که در آخر وقتی بخواهیم هم از الگوی Unit Of Work هم از Repository و هم از Generic Repository بخواهیم استفاده کنیم بایستی چه روندی را در پیش بگیریم.

خب ابتدا از اینترفیس های Repository شروع می کنیم.

این اینترفیس ها Scope را مشخصی می کردند که به طور مثال این Enlity فقط بایستی Delete یا Add را داشته باشد.

نکته : در هر Repository اگر متد SaveChanges وجود دارد، بایستی آن را حذف کرد.

خب بریم سراغ IUnitOfWork

این اینترفیس متدی با نام SaveChanges داشت و تنها وظیفه ی آن همین متد می باشد ، پس IUnitOfWork فقط بایستی متد SaveChanges در آن تعریف شده باشد و مابقی موارد بایستی حذف گردند.

خب بریم سراغ Generic Repository

این الگو از تکرار کد جلوگیری می کند، اما در صورت استفاده از این الگو ما مدیریت Scope هر Entity را از دست می دادیم ولی می توانیم آن را به صورتی هندل کنیم .

ابتدا بایستی IGenericRepository را به همراه رفرنس هایش کاملاً پاک کرد، چون مورد اضافی ای هست.

خب بریم سراغ استفاده از Generic Repository

```
public class OrderRepository(AppDbContext dbContext) : GenericRepository<Order>(dbContext),
IOrderRepository
{
    public async Task<Order> FindAsync(int id) => await dbContext.Order.FindAsync(id);
}
```

این context همان کانتکستی هست که در کلاس GenericRepository به صورت protected تعریف شده است.

ما در Order Repsitory آمدیم و از کلاس Generic Repository ارث بری کردیم و از آنجایی که این کلاس متدهای Add , Delete Update , و Get را پیاده سازی کرده است، کلاس Order Repository هم چون از این کلاس ارث بری کرده است تمامی این متدها را دارد، بعلاوه ی متدی که در خود IOrderRepository تعریف شده است.

خب ما که مجدداً Scope آن Entity را از دست دادیم ! اما این طور نیست ! چرا ؟

چون ما IOrderRepository را اینجکت میکنیم و از متدهای آن استفاده میکنم و اگر بخواهیم از هر متدی که در کلاس GenericRepository پیاده سازی شده است استفاده کنیم، بایستی Signature آن متد مربوطه در Generic Repository را درون IOrderRepository تعریف نماییم ، پس هم چنان Scope هر Entity قابل مدیریت است.

خب در آخر ما بایستی به یک سوال پاسخ بدهیم که آیا نیاز است که اصلا از Repository، UnitOfWork و Generic Repository و یا حتی استفاده از Specification Pattern را داشتند باشیم ؟ اگر بخواهیم داشته باشیم چه زمانی بایستی داشته باشیم ؟

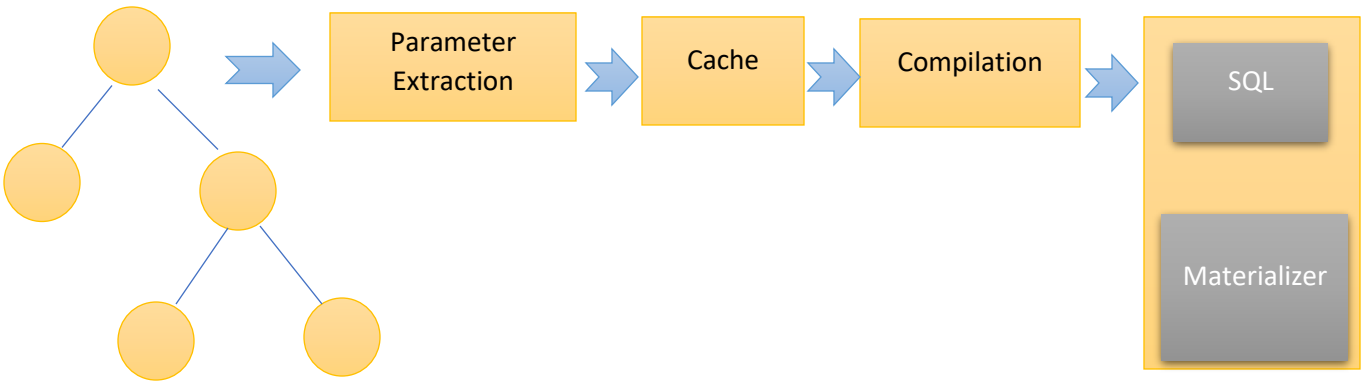
خب ابتدا بایستی از خودمان بپرسیم که آیا هم چین اتفاقی خواهد افتاد که بخواهیم ORM و یا نوع دیتابیس را تغییر دهیم ؟

اگر اتفاق خواهد افتاد، بازه زمانی آن به چه صورت خواهد بود ؟ اگر اتفاق خواهد افتاد پس ما را به سمت استفاده از الگوهایی که تا الان به آن ها اشاره کردیم می برد، اما اگر این اتفاق نخواهد افتاد می توانیم از همان روش اولی که در ابتدا شرح داده شد استفاده کرد (روشی که DbSet در اینترفیس IAppContext وجود داشت)

دیگر در این سناریو مشکلی ندارد که DbSet در این اینترفیس وجود داشته باشد و پکیج EFCore در لایه Application نصب شده باشد.

بحث Lambda To TSql Translation

مراحل اجرا و ترجمه شدن دستورات Lambda و در نهایت Materialize شدن آبجکت




```
var list =
  context.Users
  .Where(x => x.Id > 100)
  .OrderBy(x => x.Title)
  .ToList();
```

این یک Expression هست که EF با Visitor های مختلف TSql نهایی را می سازد، اما این فاز ترجمه هزینه بر و زمان بر هست پس بایستی سناریویی را در نظر گرفت که هر سری این فاز اجرا نگردد ، به همین خاطر برای جلوگیری از اجرا شدن تکراری ترجمه Lambda به TSqlسناریوی Cache توسط EF صورت گرفت و EF با توجه به Expression ورودی یک کلید می سازد و در نهایت بعد از ترجمه Expressionبه TSql آن را با کلید تولید شده Cache می کند تا در مراجعات بعدی فاز ترجمه اجرا نگرد ، اما در اینجا یک چالشی هست که برای درک بهتر آن به مثال زیر توجه کنید.

مثال اول:

```
context.Users
.Where(x => x.Id > 100);
```

مثال دوم:

```
int id = 100;
context.Users
.Where(x => x.Id > id);
```

این دو مثال از نظر ما برنامه نویسان یکسان هست اما باید توجه کرد که Expression تولید شده برای این دو یکسان نیست، پس در این صورت Compile تکراری صورت می گیرد ، برای حل این مشکل بایستی چه سناریویی در نظر گرفت ؟!

برای حل این مشکل EF قبل از اینکه Cache را چک کند ، یک فاز Parameter Extraction را در نظر گرفته تا این مشکل را بتواند برطرف کند.

اگر به طور مثال کلید Cache به شکل زیر باشد

```
Cache Key => "x => x.Id > 100"
```

بعد از در نظر گرفتن Parameter Extraction دیگر مقدار عدد 100 به صورت عددی در کلید Cache قرار نمی گیرد و کلید را به شکل زیر تولید می کند.

```
Cache Key => "x => x.Id > @Id"
```

فاز (Ling To T-SOL) Compilation in EF Internal

فاز Pre Processor : فازی که قبل از فاز ترجمه به TSql اجرا می گردد و یکی از کارهایی که در این فاز صورت می گیرد به طور مثال کوئری زیر را در نظر بگیرید.

```
var users =

content.Users
.Where(x.Id > 4 && x.name != null)
.ToList();
```

خب EF دراین فاز یک سری Visitor پیاده سازی کرده که یک سری بردسی ها را انجام می دهد و به طور مثال چک می کند که فیلد Name آیا Null پذیر هست یا خیر، که اگر Null پذیر نبود پس فیلد Name مقدار Null ندارد، پس چک کردن آن بیهوده بوده و کوئری ما را تبدیل به کوئری زیر می کند .

```
var users =
content.Users
.Where(x.Id > 4 && true)
.ToList();
```

در ویزیتور بعدی جاهایی که یک طرف شرط هست و طرف دیگر آن true هست را پاک می کند و در نهایت کوئری ما به شکل زیر می شود .

```
var users =
content.Users
.Where(x.Id > 4)
.ToList();
```

در واقع فاز PreProcessor ، فاز Optimization هست که کوئری نهایی بهینه باشد.

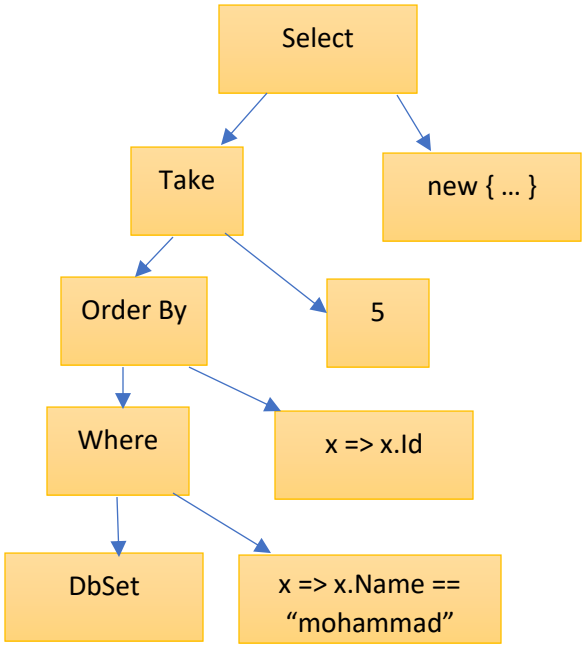
فاز **Translate**

بعد از اینکه در فاز اول بهینه سازی صورت گرفت این فاز اجرا می گردد ، برای درک بهتر این فاز به شکل زیر توجه فرمایید.

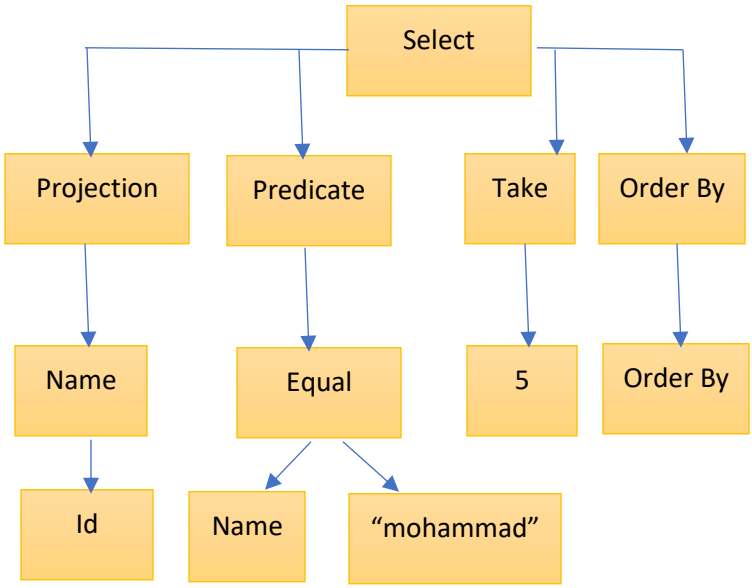
```
var users =
 context.Users
 .Where(x => x.Name == “mohammad”)
 .OrderBy(x => x.Id)
 .Take(5)
 .Select(x => new { x.Id, x.Name });
```

در ادامه می خواهیم Expression تولید شده قابل فهم برای Linq و SQL را بررسی نماییم.

Expression LINQ World



Expression SQL World

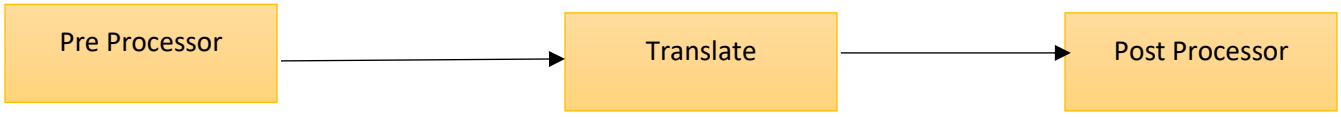


فاز PostProcessor

در این فاز Expression نهایی که قابل فهم برای دنیای SQL هست را گرفته و تبدیل به TSql نهایی می کند که در واقع یک Visitor هست به نام QuerySqlGenerator که با توجه به Provider مربوطه کوئری نهایی را می سازد.

نکته : نکته ای که بایستی در نظر داشت این هست که به طور مثال Take(5) در Sql Server برای واکشی N رکورد دلخواه از Top 5 اما در Postgres دستور Limit 5 هست که در فاز PostProcessor ترجمه کوئری نهایی با توجه به Provider نهایی انجام می گیرد.

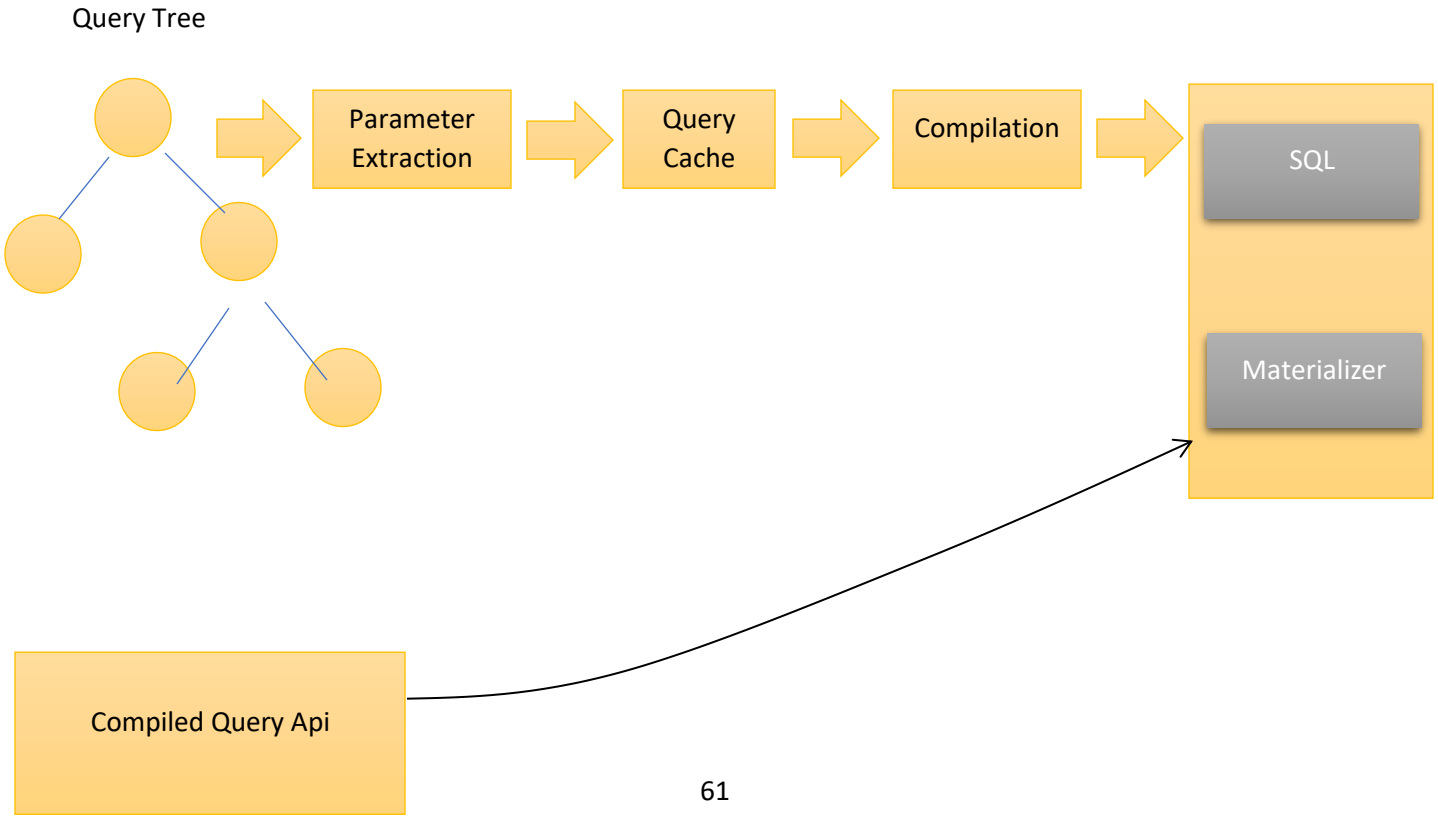
پس در کل ما سه فاز زیر را داریم که شرح هر یک را دادیم.



نکته : در سه فاز بالا کلی Visitor پیاده سازی شده است و به همین خاطر فاز Compilation کند هست و برای آن Cache در نظر گرفته شده است.

بحث Compiled Query Api

در حالت عادی ما سناریوی زیر را داریم.



اما EF امکانی را برای ما فراهم کرده تا بتوانیم یک بار آن را کامپایل و N بار آن را اجرا کنیم.

سناریوی زیر را در نظر بگیرید :

```
While(true)
{
  var query =
   context.Tickets.
  Where(x => x.State ==   State.Finish)
  .ToList;()

  query.Foreach(x  =>
  {
     // Send Sms
  )};
  Thread.Sleep;(5000)
}
```

ما در سناریوی بالا هر پنج ثانیه در حال دریافت اطلاعات از دیتابیس هستیم ، خب در این سناریو کوئری ما مدام بایستی فاز Parameter Extraction صورت گیرد ، Cache چک شود و .... خب این کارها باعث کندی می شود که در ادامه به رفع این مشکل می پردازیم.

یکی از قابلیت هایی که EF در اختیارمان قرار داده این هست که ما بتوانیم یک بار کوئری را Compile کرده و n بار بدون اینکه Cache چک شود و یا Parameter Extraction صورت گیرد مستقیما کوئری اجرا شود ، برای این کار کوئری قبلی را بایستی به روش زیر نوشت.

```
var compiledQuery = EF.CompileQuery((AppContext context) =>
                        content.Tickets.Where(x => x.state == State.Finish);

while(true)
{
    var query = compiledQuery(context).ToList;()
    query.Foreach (x  =>
    {
        // Send Sms
     )};
    Thread.Sleep;(5000)
}
```

در این روش کوئری فقط یکبار ترجمه می شود و در دفعات دوم به بعد دیگر مستقیما اجرا می شود و افزایش سرعت زیادی را مشاهده خواهیم کرد.

بحث Identity Resolution

در Change Tracker کلاس ها یا همان Net Object. ها را با PK مربوطه Detect و Track می کند، اگر یک Object را با آیدی 4 Track کند و بخواهد مجدداً یک Object دیگری را با آیدی 4 Track کند Exception رخ می دهد که به این موضوع Identity Resolution می گویند.

بحث AsNoTracking

دو تا کار را برای ما انجام می دهد، یک اینکه Change Tracker را متوجه Track نکردن Object مورد نظر می کند و دو اینکه باعث در نظر نگرفتن Identity Resolution می شود. (در این سناریو رفرنس هر دو آبجکت یکسان نیست)

بحث AsNoTrackingWithIdentityResolution

زمانی که می خواهیم آبجکت ما Track نشود اما Identity Resolution چک شود و در رکورد با PK یکسان نداشته باشیم از این متد استفاده می کنیم. (در این سناریو ما دو رکورد داریم که رفرنس یکسان دارند)

بحث Transaction

نکته : در EF متد SaveChanges به صورت پیش فرض Transactional عمل می کند و اگر در زمان اجرای این متد خطایی رخ دهد ، هیچ یک از تغییرات اعمال نخواهد شد.

یک مثال از Transaction

```
using var transaction = context.Database.BeginTransaction;()

try
{
    //some sose
    context.SaveChange();
   transaction.Commit();
}
catch
{
  transaction.Rollback();
}
```

استراتژی **Retry** در **Transaction** ها

```
using var transaction = context.Database.BeginTransaction();
var auther = new Auther { Name = "Hamid" };
var post = new Post { Title = "C# Book" };

try
{
  context.Authers.add(auther);
  context.SaveChanges();                    Step 1

  transaction.CreateSavePoint("Auther_Added");

  post.AutherId = auther.Id;
  context.Posts.Add(post);                  Step 2
  context.SaveChanges();

  transaction.Commit();
}
catch
{
     post.Title = "C#";
    transaction.RoolBackToSavePoint("Auther_Added");
    context.Posts.Add(post);
    context.SaveChanges();

    transaction.Commit();
}
```

توضیحات کدهای فوق

ما در کد فوق احتمال می دهیم که Step 1 با موفقیت انجام می شود و هم چنین احتمال می دهیم که در Step 2 خطایی رخ می دهد ، اما نمی خواهیم در صورت رخداد خطا کل عملیات Rollback شود.

پس بعد از Step 1 یک Point ایجاد می کنیم و زمانی که خطا رخ داد ، احتمالا دلیل خطا زیاد بودن تعداد کاراکترهای Title باشد ، برای همین زمانی که خطا رخ داد تعداد کاراکترها را کم تر کرده و Transaction را به یک گام بعد از SaveChanges در Step 1 هدایت می کنیم تا ادامه کار مجددا اجرا شود.

در ادامه بحث Transaction می خواهیم به سناریویی اشاره کنیم که بنا بر دلایلی ما به دو نمونه از Context نیاز داریم و اینکه چطور می توانیم در این دو نمونه از یک نمونه Transaction استفاده کنیم و اصطلاحا Transaction را Share کنیم.

برای درک بهتر موضوع به مثال زیر توجه کنید.

```
using var context1 = new AppContext();
using var transaction = context1.DatabaseBeginTransaction();
try
{
    var auther = new Auther { Name = "Hamid" };
    context1.Auther.Add(auther);
    context1.SaveChanges();

    using var context2 = new AppContext();
    context2.Database.UseTransacrion(transaction.GetDbTransaction());
    var post = new Post { AutherId = auther.Id, Title = "C#" };
    contect2.Post.Add(post);
    contect2.SaveChanges();

    transaction.Commit();
}
catch
{
    transaction.Rollback();
}
```

در مثال فوق ما توانستیم توسط متد Use Transaction از یک Transaction قبلی استفاده کنیم و تغییرات دو Context را در یک Commit اعمال نماییم.

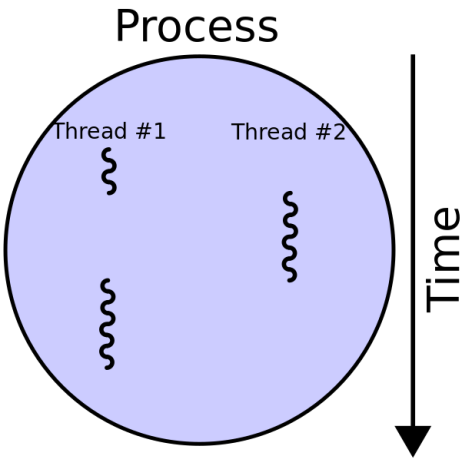
در نوع بعدی از Transaction ها ما می توانیم از External Transaction هم استفاده کنیم که در فضای نام System.Transaction موجود است و خوشبختانه EF توانسته کاملا خودش را با آن تطبیق دهد.

برای درک بهتر موضوع به مثال زیر توجه فرمایید.

```
using (var scope = new TransactionScope(
TransactionScopeOption.Required,
new TransactionOptions { OsolationLevel = IsolationLevel.ReadCommitted }))
{
  try
  {
     using var context = new AppContext();
     var aurher = new Auther { Name = "Hamid" };
     context.Auther.Add(auther);
     context.SaveChanges();

     var post = new Post { AutherId = auther.Id , Title = "C#" };
     context.Post.Ads(poat);
     context.SaveChanges();

     scope.Complete();
  }
   catch
  {
        //Do some code such as log and....
  }
}
```

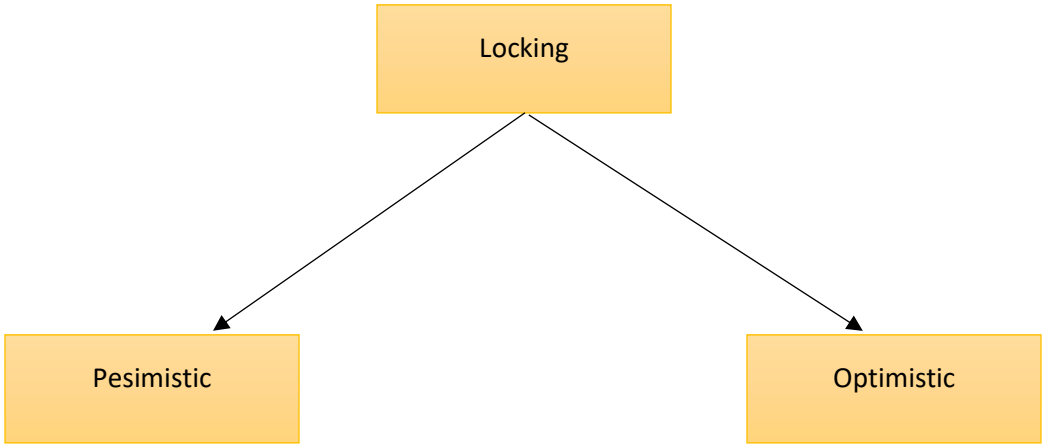
بحث Handling Transacrion

نکته : هم زمانی یعنی اجرای دو یا چند Transaction در یک زمان.



ما برای افزایش پرفرمنس از Concurrency استفاده می کنیم ، چون نمی توانیم درخواست ها را به صورت سریالی اجرا کنیم ، اما زمانی که از آن استفاده می کنیم احتمال وقوع Conflict نیز هست.

یکی از راه های هندل کردن Concurrency استفاده از Lock هست و ما دو نوع Lock داریم که در ادامه به آن ها اشاره خواهیم کرد.



Lock مستقیم بر روی خود رکورد اعمال می کند.

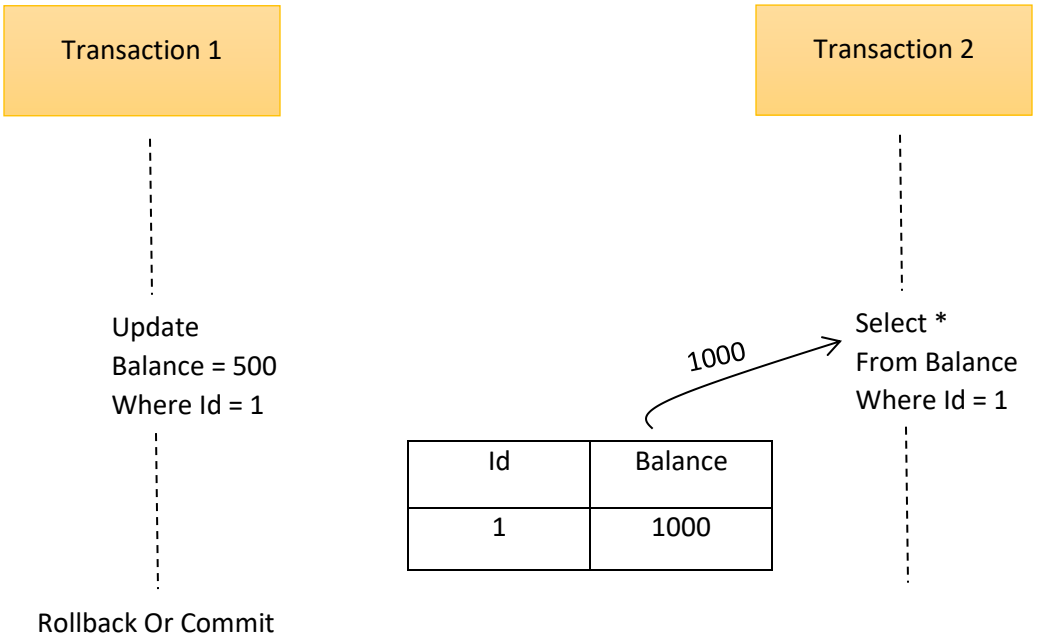
مانند استفاده از Row Versioning

بحث هم زمانی در فضای **Read Data**

بحث **Read UnCommitted**

ما یک اصطلاحی داریم تحت عنوان Dirty Data که در فضای Read اتفاق می افتد ، اما چطور ؟

برای درک بهتر موضوع به مثال زیر توجه فرمایید.



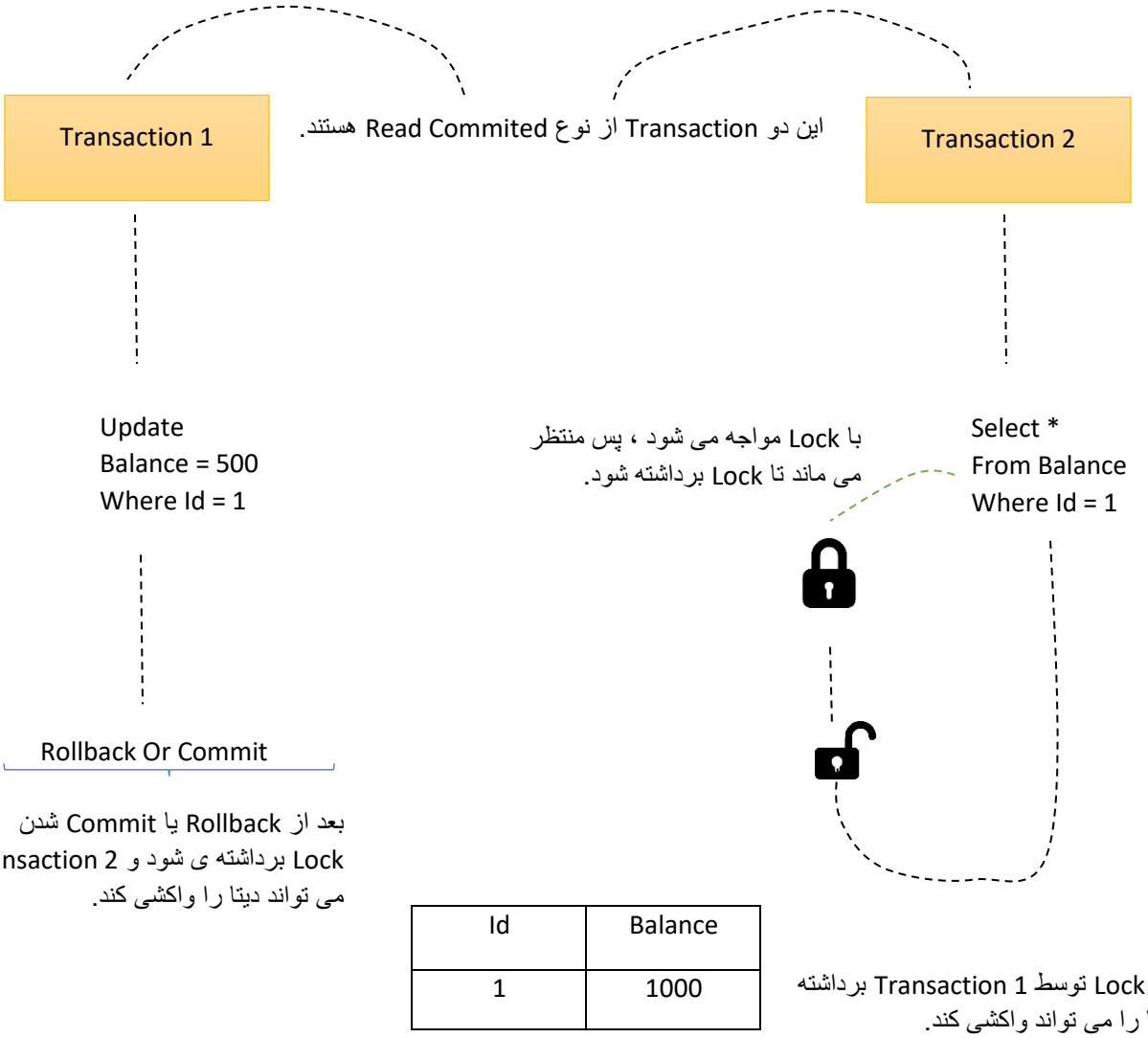
در مثال فوق ما در Transacrion 1 اگر تغییری که در Balance می دهیم Commit شود، Balance ما برابر با 500 می شود اما Transaction 2 مقدار 1000 را برگردانده است و اگر هم Commit نشود که همان مقدار باقی می ماند که به این موضوع Dirty Data می گویند.

نکته : بین این دو Transaction هیچ Isolation Level وجود ندارد و به همین خاطر هیچ کنترلی روی یکدیگر ندارند.

نکته : در مثال بالا بالاترین سطح زمانی یعنی Read UnCommitted وجود دارد و دیتا را حتی اگر Commit نشده باشد ، آن را واکشی می کند که دقیقا همین اتفاق باعث وجود Dirty Data می شود.

در سطح Read UnCommitted دیدیم که Dirty Data وجود دارد ، خب بایستی قدری سخت گیرانه تر عمل کنیم که در ادامه به آن اشاره می خواهیم کرد.

بحث Read Committed



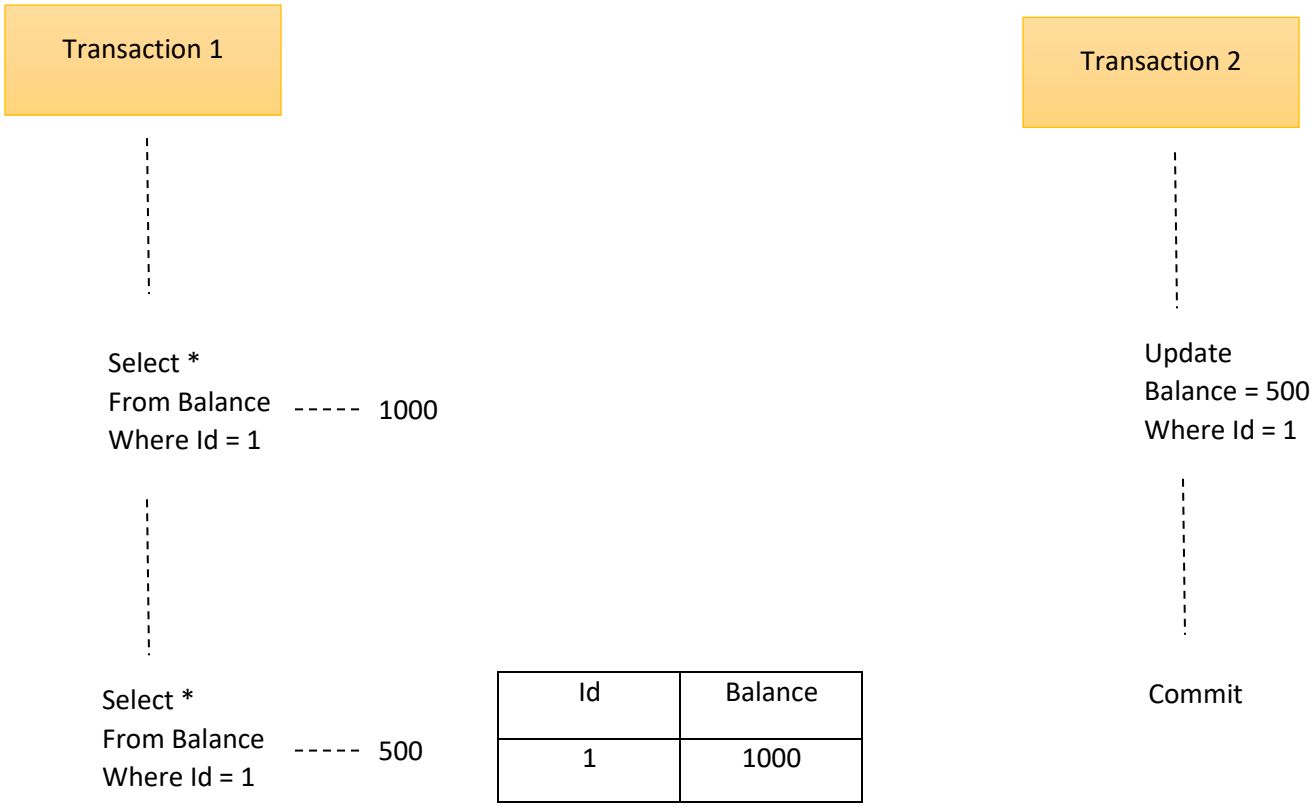
نکته : در سناریوی فوق دیگر Dirty Data وجود ندارد.

نکته : پیش فرض دیتابیس هم حالت Read Committed هست.

اما این سطح هم بدون ایراد نیست که در ادامه به مشکل آن اشاره خواهیم کرد.

در ادامه ی بحث Read Committed بایستی به مشکل Repeatable Read اشاره کرد، اما Repeatable Read یعنی چی ؟

برای درک بهتر موضوع به مثال زیر توجه فرمایید.



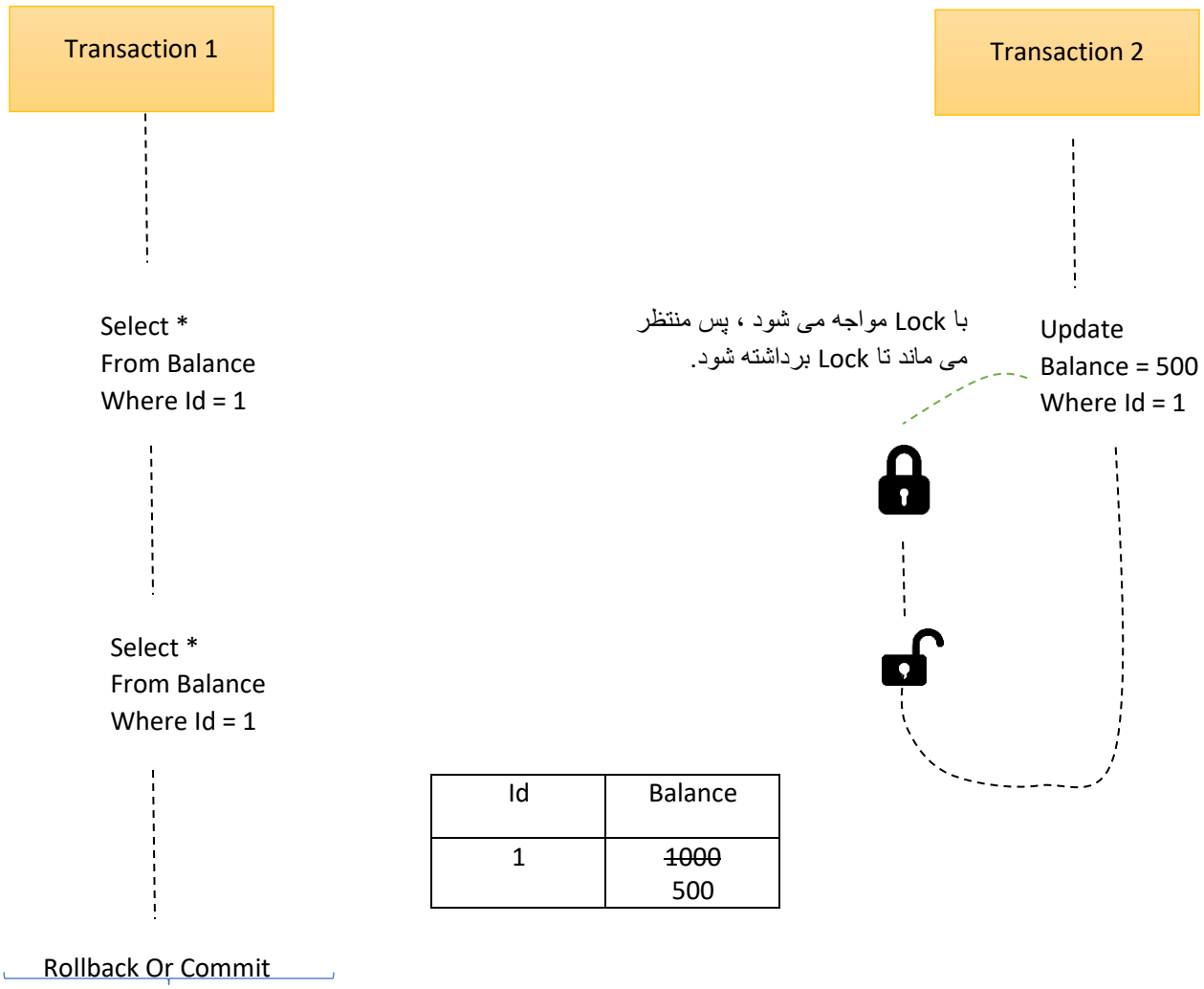
در مثال فوق در Transaction 1 ابتدا Id با مقدار یک واکشی شده با مقدار بالانس 1000 در این حین Transaction 2 مقدار بالانس همین رکورد را می خواهد تغییر دهد و مقدار بالانس را از 1000 به 500 تغییر می دهد و در اینجا چون Lock وجود ندارد، پس بدون هیچ وقفه ای تغییرات Commit می شود و در ادامه Transaction 1 مجددا همین رکورد را واکشی می کند اما با چه مقدار بالانسی ؟!

در اینجا مقدار بالانس تغییر کرده که در واکشی اول مقدار بالانس 1000 بوده و الان به مقدار 500 تغییر کرده است.

در اینجا بایستی مجددا سخت گیرانه تر عمل کرد و از Isolation Level دیگری استفاده کرد.

نکته: برای اینکه بتوانیم مشکل Repeatable Read را حل کنیم، بایستی از isolationLevel با نام Repeatable Read استفاده کرد.

برای درک بهتر Repeatable Read به مثال زیر توجه فرمایید.

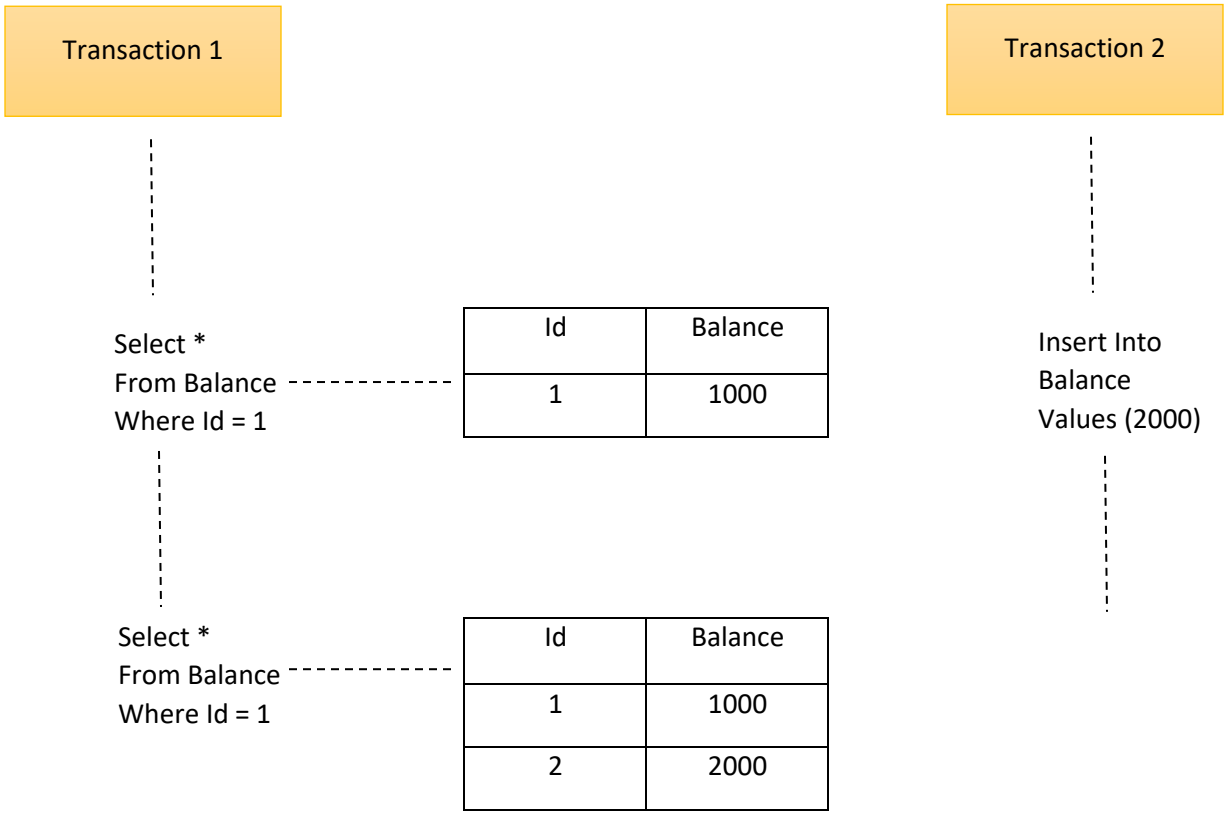


Id	Balance
1	~~1000~~ 500

در مثال فوق مشاهده می کنیم که در دو Select مقدار Balance برابر با 1000 است و زمانی که Transaction 2 می خواهد عملیات Update انجام دهد با Lock مواجه می شود و تا زمانی که کار Transaction 1 به اتمام نرسد نمی تواند عملیات Update را انجام دهد.

نکته : در این سناریو خبری از Dirty Data و یا Repeatable Read نیست.

در ادامه بحث Repeatable Read بایستی گفت که این Isolation Level هم بدون ایراد نیست و مشکلی که دارد ، مشکل Phantom هست که برای درک بهتر آن به مثال زیر توجه فرمایید.



ما در سناریوهای قبل از هم زمانی در زمان Read و Update صحبت کردیم ، اما صحبتی از Insert نکردیم.

در Transaction 1 و در اولین Select ما تنها یک رکورد را واکشی کردیم اما در این حین Transaction 2 قصد درج یک رکورد را دارد و در ادامه ی Transacrion 1 جایی که دومین Select اجرا می شود ، با دو رکورد مواجه می شود که در اینجا با قضیه ی Phantom مواجه می شویم و برای حل این مشکل بایستی از Isolation Level با نام Serializable استفاده کرد.

در اینجا قصد داریم یک مرور کلی بر روی Isolation Level هایی که تا الان به آن ها اشاره کردیم داشته باشیم و به مزایا و معایب هر یک اشاره کنیم .

نکته : هر چه ما به سمت Serializable نزدیک تر می شویم Concurrency کم تر شده و Dirty Data نیز کم تر خواهد شد.

تا به اینجای کار ما در مورد Pesimistic ها صحبت کردیم اما از الان به بعد می خواهیم در مورد Optimistic ها صحبت کنیم که در این زمینه EF خیلی به ما کمک خواهد کرد.

بریم با هم در مورد اولین مورد از Optimostic ها را صحبت کنیم.

بحث RowVersioning

مدیریت این نوع از مدیریت هم زمانی به عهده ی دیتابیس هست که در زمان Insert و یا Update یک ورژن جدید را مقدار دهی می کند.

برای استفاده از RowVersioning می توانیم به شکل زیر عمل کنیم.

کافیست در Entity مربوطه پراپرتی زیر را اضافه کنیم.

```
[Timestamp]
public byte[] RowVersion { get; set ; }
```

نکته : تنها با اضافه کردن پراپرتی بالا اگر هر یک از مقادیر رکورد مربوطه تغییر کند ، Version جدیدی برایش مقدار دهی می شود.

نکته : اگر بخواهیم از اتربیوت استفاده نکنیم و از Fluent برای کانفیگ استفاده کنیم ، می توانیم به صورت زیر عمل کنیم .

```
builde.Entity<Account>()
.Property(x => x.RowVersion)
.IsRowVersion();
```

در سناریوی قبلی که از RowVersion استفاده کردیم یک مشکلی داشت که شاید ما نخواهیم با تغییر هر مقداری از رکورد Version جدیدی برای آن مقدار دهی شود ، مثلا اگر Title تغییر کرد خیلی برایمان اهمیت نداشته باشد ولی اگر مقدار Amount تغییر کرد اهمیت داشته باشد و هم زمانی چک شود.
برای استفاده از این سناریو می توانیم به شکل زیر عمل کنیم.

```
[ConcurrencyCheck]
public Guid CToken { get; set; }
```

با اضافه کردن اتربیوت ConCurrencyCheck در زمانی که مقدار Amount تغییر کرد ، بایستی خودمان در سطح Application مقدار CToken را تغییر دهیم.

اگر بخواهیم Optimistic را به صورت کلی و در یک نگاه داشته باشیم ، می توانیم آن را به صورت زیر داشته باشیم .

1 - RowVersion
- مدیریت آن بر عهده ی دیتابیس می باشد
- تولید مقدار Version جدید با تغییر هر یک از مقادیر رکورد

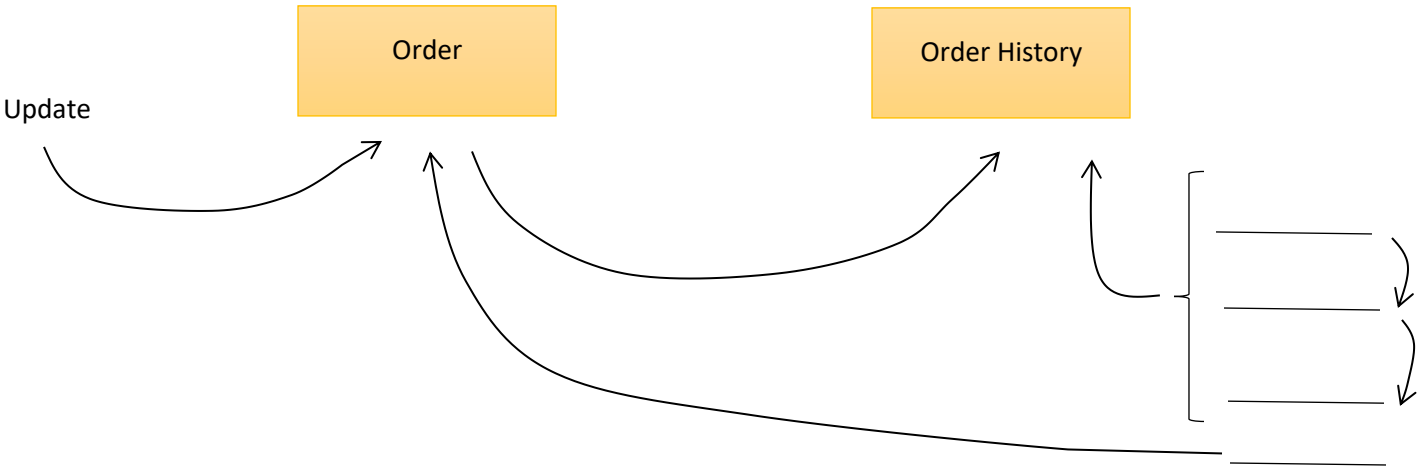
2 - ConCurrencyCheck
- مدیریت آن بر عهده ی خودمان می باشد
- در صورت نیاز توسط خودمان و در سطح Application مقدار جدیدی به آن داده می شود

نکته : اگر نخواهیم از اتربیوت برای کانفیگ ConCurrencyCheck استفاده کنیم و آن را با Fluent کانفیگ کنیم ، می توانیم به صورت زیر عمل کنیم.

```
Builder.Entity<Account>()
.Property(x => x.Ctoken)
.IsConCurrencyCheck();
```

بحث Temporal Table

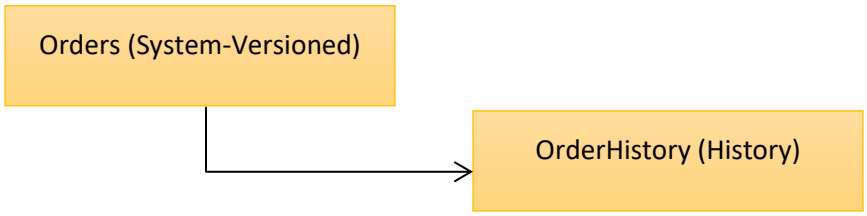
اگر بخواهیم تاریخچه ی یک جدول را داشته باشیم ، به طور مثال رکورد اول با آیدی یک و Amount با مقدار 1000 درج شده است و سپس مقدار Amount آن ویرایش شده و شده 1600 و .... که سیر ویرایش یک رکورد را خواهیم داشت .
برای داشته تاریخچه ی یک جدول اولین چیزی که به ذهنمان خطور می کند این هست که خودمان بیاییم که جدول History داشته باشیم و آن را مدیریت کنیم.
که روند مدیریت آن به شکل زیر می باشد.



در مثال فوق در زمان Insert هیچ اتفاقی نمی افتد ، اما در زمان Update رکورد مربوطه قبل از اینکه ویرایش گردد در جدول History درج می گردد و سپس ویرایش می شود.
همه ی این کار ها را خودمان بایستی مدیریت کنیم ، اما چه خوب می شد قابلیتی بود که این مدیریت را بر عهده می گرفت و کارمان را راحت تر می کرد.
بریم با هم در ادامه در مورد این ویژگی صحبت کنیم.
قابلیتی که مدیریت History را بر عهده می گیرد Temporal Table است.
در این قابلیت ما تنها با جدول Order کار می کنیم و کاری با جدول History نخواهیم داشت ، اما EF یک سری API در اختیار ما قرار می دهد تا بتوانیم در صورت نیاز روی جدول History نیز دیتاهایی را واکشی کنیم.
بریم با هم ببینیم چطور می توانیم از این قابلیت در EF استفاده کنیم.
برای استفاده تنها کافیست در متد OnModelCreating کانفیگ زیر را اعمال کنیم.

```
builder.Entity<Order>()
.ToTable("Orders", t => t.IsTemporalTable());
```

زمانی که از این قابلیت استفاده می کنیم در SSMS در زیر کنار نام جدول (Syatem-Versioned) ذکر شده و در زیر مجموعه ی آن یک جدول هم نام برای History وجود دارد .



نکته : زمانی که از قابلیت Temporal Table استفاده می کنیم دو Shaddow Property به جدول مربوطه اضافه می گردد که در سمت Entity وجود ندارد اما در سمت دیتابیس وجود دارد.

دو Shadow Property با نام های

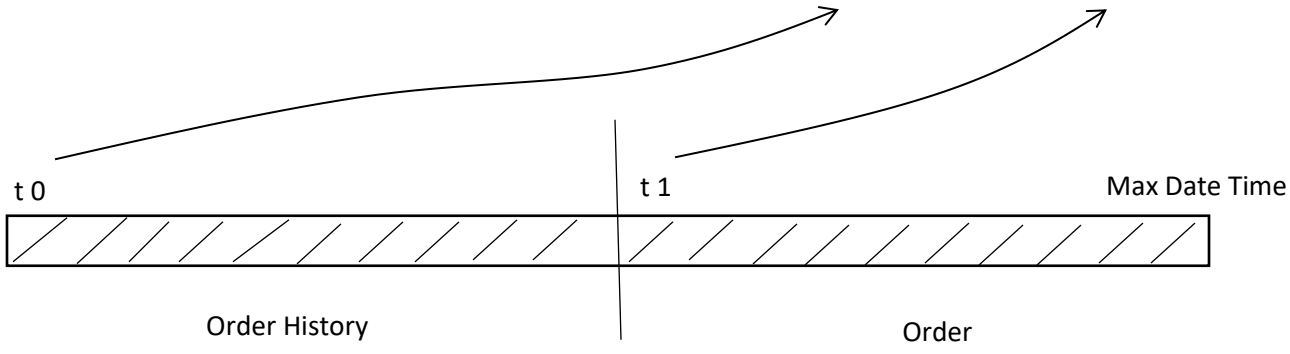
- Period Start

- Period End

که مشخص می کنند این رکورد از چه تاریخی و تا چه تاریخی فعال بوده اند.

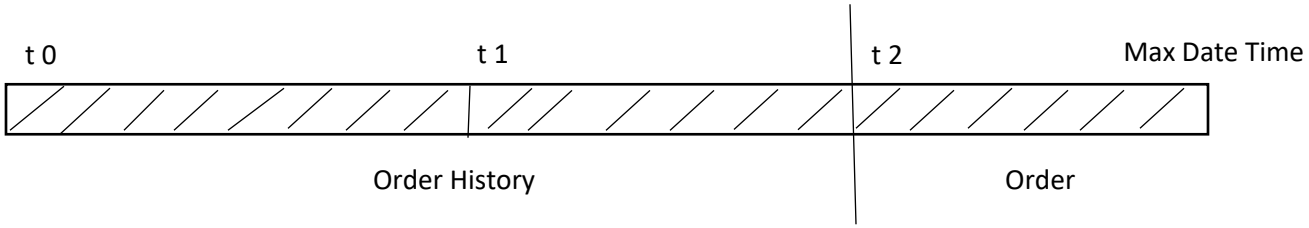
Order History

Id	Number	TotalPrice	PeriodStart	PeriodEnd
1	1000	200000	2024-11-10	2024-12-01



حالا اگر یک Update دیگری صورت گیرد ، دیتای زیر را خواهیم داشت.

Id	Number	TotalPrice	PeriodStart	PeriodEnd
1	1000	200000	2024-11-10	2024-12-01
2	1001	250000	2024-11-13	2024-12-03



در ادامه بحث Temporal Table ها EF این امکان را به ما می دهد تا بتوانیم نام دو Shadow Property و یا حتی نام جدول History را هم عوض کنیم که برای انجام این کار می توانیم به صورت زیر عمل کنیم.

```
builder.Entity<Order>()
.ToTable("Orders", t => t.IsTemporalTable(x  =>
{
  x.HasPeriodStart("ValidFrom");
  x.HasPeriodEnd("ValidTo");
  x.UseHistoryTable("OrderAudit");
}));
```

برای کار با Temporal Table ها EF پنج متد در اختیارمان قرار می دهد که در ادامه به هر یک از آن اشاره خواهیم کرد.

۱ - Temporal All

اگر بخواهیم تمامی History مرتبط با هر رکورد را واکشی کنیم از این متد بایستی استفاده کنیم.

برای درک بهتر متد و نحوه استفاده از آن به مثال زیر توجه فرمایید.

```
var history = context
.Orders
.TemporalAll()
.Where(x => x.Id == 1)
 .OrderBy(x => EF.Property<DateTime>(x, "ValidFrom"))
.Select(x => new
{
   Order = x,
   ValidFrom = EF.Property<DateTime>(x, "ValidFrom"),
    ValidTo = EF.Property<DateTime>(x, "ValidTo")
}).ToList();
```

۲ - TemporalFromTo

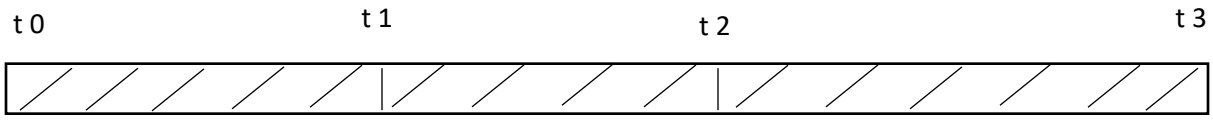
اگر بخواهیم دیتاهای یک بازه زمانی را واکشی کنیم از این متد می توانیم استفاده کنیم.

```
var from = DateTime.Parse("07:49:59.4159260 2024-04-07");
var to = DateTime.UtcNow;

var history = context
.Orders
.TemporalFromTo(from, to);
.Where(x => x.Id == 1)
.OrderBy(x => EF.Property<DateTime>(x, "ValidFrom"))
.Select(x => new
{
   Order = x,
   ValidFrom = EF.Property<DateTime>(x, "ValidFrom"),
   ValidTo = EF.Property<DateTime>(x, "ValidTo")
}).ToList();
```

نکته : TemporalFromTo دو ورودی From و To دارد که حتی به نانو ثانیه آن نیز حساس است.
ما اگر فقط بازه ی تاریخی t0 تا t1 را بخواهیم واکشی کنیم بایستی مقداری که به ورودی می دهیم دقیقا بایستی تو بازه ی t0 تا t1 باشد ، اما اگر حتی یک نانو ثانیه هم بیشتر باشد به جای t0 تا t1 ، بازه ی t0 تا t2 را واکشی می کند.

۳ - TemporalBetween

این متد همانند متد TemporalFromTo هست ، اما با یک سری تفاوت ها که برای درک بهتر این تفاوت به مثال زیر توجه فرمایید.

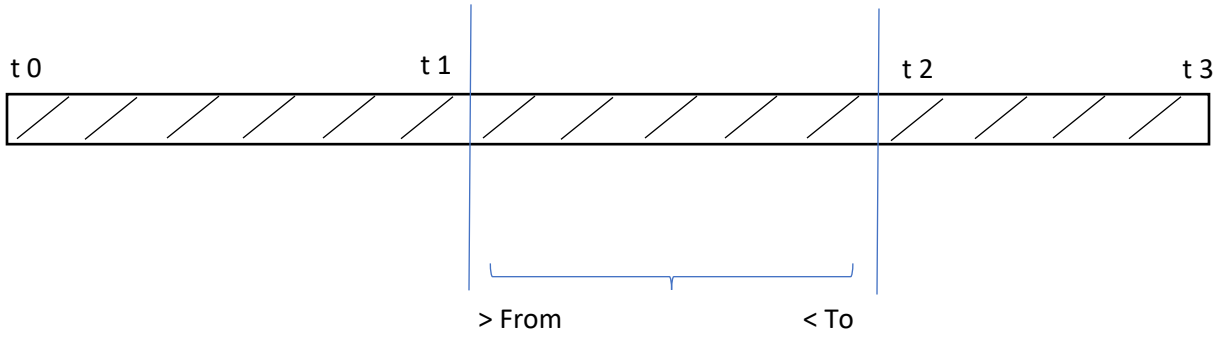


temporalFromto ➔	>= From	< To
Temporalbetween ➔	>= From	<= To

در مثال بالا اگر بخواهیم از بازه ی t0 تا t2 را در نظر بگیریم ، TemporalFromTo دو رکورد را واکشی می کند ولی TemporalBetween سه رکورد را بر می گرداند.

۴ - TemporalContainedIn

دیتاها را در دو بازه ی زمانی مشخص شده واکشی می کند.




```
var from = DateTime.Parse("07:49:59.4159260 2024-04-07");
var to = DateTime.UtcNow;

var history = context
.Orders
.TemporalContainedIn(from, to)
.Where(x => x.Id == 1)
.OrderBy(x => EF.Property<DateTime>(x, "ValidFrom"))
.Select(x => new
{
  Order = x,
  ValidFrom = EF.Property<DateTime>(x, "ValidFrom"),
  ValidTo = EF.Property<DateTime>(x, "ValidTo")
}).ToList();
```

۵ - TemporalAsOf

یک Timestamp از ورودی دریافت می کند و بر اساس ValidFrom آن یک رکورد را واکشی می کند .

```
var time = DateTime.Parse("2024-04-07 08:16:43.4577641");
var orderHistory = context
.Orders
.TemporalAsOf(time)
.FirstOrDefault(x => x.Id == 1);
```

تا اینجا ما پنج متدی که EF در اختیار ما می گذراد برای کار با Temporal Table ها را شرح دادیم و برای هر یک مثالی زدیم.