عنوان مقاله: بررسی کامل Complex Type در EF Core ۸ همراه با مثال های عملی

نویسنده مقاله: تیم فنی نیک‌آموز

تاریخ انتشار: ۱۷ اسفند ۱۴۰۲

منبع: https://nikamooz.com/complex-type-in-ef-core-۸/

Complex Type در EF Core 8 چه نوع داده‌ای است و چه تفاوتی با سایر تایپ‌ها دارد؟ در این مقاله، قصد داریم به چیستی این Type پرداخته و اهمیت آن را به‌صورت عملی شرح دهیم.

Data Type هایی همچون Bool, String, Int و غیره که توسط کامپایلر ازپیش شناخته‌شده هستند، تایپ‌های اولیه یا Primitive Type نام دارند. برای خواندن و نوشتن این تایپ‌ها در جداول پایگاه داده، به پیش‌نیاز یا پردازش ویژه‌ای احتیاج نیست؛ زیرا که تمامی دیتابیس‌ها این نوع از دیتاتایپ‌ها را می‌شناسند و از آن‌ها برای ذخیره دیتا استفاده می‌کنند. در نقطه مقابل، تایپ‌هایی وجود دارند که به‌صورت سفارشی ایجاد می‌شوند و برای ذخیره دیتا مورد استفاده قرار می‌گیرند. سؤال مهم این است که چرا در دنیای برنامه‌ها، گاهی باید به تعریف تایپ‌هایی بپردازیم که توسط **پایگاه داده** شناخته‌شده نیستند؟ برای درک چرایی این موضوع، به مثال زیر توجه کنید:

فرض کنید متدی داریم که سه ورودی از نوع String دریافت می‌کند. نخستین ورودی، نام، بعدی نام خانوادگی و آخری نیز ایمیل کاربر است:

```
public void Register(string name, string family, string email)
{
    // Do somethings
}
```

حال می‌خواهیم از این متد برای ثبت نام کاربر استفاده کنیم. احتمال این که به جای نام، اشتباهی ایمیل کاربر را به ورودی متد بدهید، چقدر است؟ هر سه پارامتر از نوع String هستند و اشتباهی جای یکدیگر قابل استفاده خواهند بود. اکنون فرض کنید اولی از جنس یک کلاس به نام Name، دومی از جنس کلاسی به اسم Family و سومی از جنس یک کلاس به نام Email باشد؛ آیا این اشتباه رخ می داد؟ پاسخ این پرسش خیر است.

برای درک بهتر، به کد زیر دقت کنید:

```
public void Register(Name name, Family family, Email email)
{
    // Do somethings
}
```

در کد بالا، این احتمال که به جای تایپ Name، تایپ Email پاس داده شود، صفر است؛ زیرا در این صورت، با خطایی هنگام کامپایل مواجه خواهید شد. کلاس‌هایی مانند Family, Name و Email تحت عنوان Strong Type اشاره می‌شود و یکی از دلایل استفاده از آن‌ها، ویژگی مذکور است. البته دلایل دیگری نیز برای اهمیت آن‌ها دارد که موضوع این مقاله محسوب نمی‌شوند. به‌عنوان مثال، چک کردن یک سری Rule های بیزینسی در داخل این کلاس‌ها یک مورد است.

در عمل، نوع Strong Type روشی برای مدل کردن Value Object ها به‌شمار می‌رود. Value Object ها شی‌هایی هستند که مقداری را ارائه می‌دهند و شناسه منحصربه‌فردی ندارند؛ این یعنی، مقدار Value Object ها مهم است. هردو یکسان هستند که اگر تمامی مقادیر Value Object به‌صورت نظیربه‌نظیر باهم برابر و یکسان باشند، می‌توان گفت آن دو یکسان هستند. در نقطه مقابل Value Object ها، دو Entity درصورتی باهم برابر هستند که شناسه‌های آن‌ها یکسان باشد.

در مثال پیشین، یک متد به نام Register به‌کار برده شد که سه ورودی از نوع Strong Type دارد. با توجه به تعریف Value Object ها، می‌توان اذعان داشت که ورودی‌های متد مذکور در اصل Value Object هستند. فرض کنید دو کلاس از نوع Name وجود داشته باشد و هردوی این کلاس‌ها، مقداری برابر با نام «پویا» دارا باشند. در چنین شرایطی، این دو کلاس با یکدیگر برابر هستند؛ زیرا مقداری که ذخیره کرده‌اند، یکسان است.

نحوه استفاده از Value Object در EF Core 8

فرض کنید تعدادی Value Object در دسترس هستند که می‌دانیم به چه منظور در سطح بیزینس و اپلیکیشن می‌شوند. حال می‌خواهیم Value Object ها را در دیتابیس استفاده کنیم. بنا به دلایلی، استفاده از آن‌ها به‌راحتی Primitive Type ها نخواهد بود. در این بخش قصد داریم به این موضوع در EF Core 8 رسیدگی کنیم. برای درک هرچه بیشتر، به سناریو زیر توجه کنید:

فرض کنیم که یک درگاه پرداخت داریم که شامل Url، Name و Payment است. برای سادگی کار، Name و Url را Primitive Type و کلاس Money را Strong Type درنظر می‌گیریم.

ابتدا یک کلاس Value Object به نام Payment تعریف خواهیم کرد که سه Property دارد. یکی از این Property ها برای نگهداری مقدار پول، دیگری برای در نظر گرفتن rate نسبت به دلار و آخرین Property برای مشخص کردن واحد پولی استفاده می‌شود.

تعریف این کلاس به شکل زیر خواهد بود:

```
public class Payment
{
    public readonly decimal Amount;
    public readonly string Currency;
    public readonly decimal Rate;

    public Money(decimal amount, string currency, decimal rate)
    {
        Amount = amount;
        Currency = currency;
        Rate = rate;
    }
}
```

کلاس Payment سه فیلد public و readonly برای نگهداری مبلغ، میزان rate و واحد پولی دارد که تنها ازطریق سازنده کلاس قابل مقداردهی خواهند بود. به بیان دیگر، می‌توان گفت که Payment یک Value Object Immutable محسوب می‌شود. این یعنی، مقدار آن تنها در زمان نمونه‌سازی قابل تعریف است و در سایر شرایط، قابل ویرایش و تغییر نخواهد بود. سؤال مهم این است که چرا یک مدل را Immutable درنظر گرفتیم؟

پاسخ این سؤال خیلی به کاربری پروژه مدنظر بستگی دارد. با این وجود، به‌صورت کلی اگر بخواهید مدلی را از دیتابیس بخوانید، نباید این امکان در آن وجود داشته باشد که مقادیر آن در طول اپلیکیشن تغییر کنند. بدین شیوه، تضمین می‌شود که داده‌های خوانده‌شده از پایگاه داده، در طول چرخه اجرای یک اپلیکیشن، دست‌نخورده و بدون مشکل باقی می‌مانند. توجه کنید این که یک مدل را Immutable درنظر بگیریم یا خیر، کاملاً مستقل از بحث Complex Type در EF Core است.

در قدم بعدی، مدل Gateway یا درگاه پرداخت را مشابه زیر ایجاد خواهیم کرد:

```
public class Gateway
{
    public long Id { get; set; }
    public required string Name { get; set; }
    public required string Url { get; set; }
    public required Payment Payment { get; set; }
}
```

کلاس Gateway یک Value Object است که Url، Name و Payment را شامل می‌شود. Property های این کلاس از بیرون قابل مقداردهی هستند، اما نسبت به کلاس Payment یک ویژگی اضافه‌تر نیز دارا هستند. تمامی Property های کلاس Gateway از نوع Required هستند؛ این یعنی، حتماً باید مقداری داشته باشند. توجه کنید که چنین مفهومی، معادل استفاده از IsNotNull در دیتابیس‌ها است.

پیش از ذخیره‌سازی مدل Gateway در دیتابیس، لازم است زیرساخت موردنیاز را فراهم کرده و مدل را به‌نحوی با EF Core کانفیگ کنیم که امکان ذخیره‌سازی آن در دیتابیس وجود داشته باشد.

دو روش برای کانفیگ این مدل‌ها در دات نت ۸ وجود دارد:

- استفاده از Mapping Attribute
- ComplexProperty API در OnModelCreating

در ابتدا با کمک دستور زیر، EF Core 8 را به‌همراه تعدادی پکیج دیگر نصب کنید:

```
dotnet add package Microsoft.EntityFrameworkCore --version 8.0.0
dotnet add package Microsoft.EntityFrameworkCore.Design --version 8.0.0
dotnet add package Microsoft.EntityFrameworkCore.Tools --version 8.0.0
dotnet add package Microsoft.EntityFrameworkCore.SqlServer --version 8.0.0
```

شایان ذکر است که Entity Framework Core پکیج اصلی برای کار با ORM قدرتمند EF Core است. برای طراحی، نیاز به پکیج Design داریم؛ EF Core از آن برای ایجاد Migration استفاده می‌کند. ضمن اینکه EF Core از یکسری CLI یا Command Line استفاده می‌کند که با نصب پکیج Tools در دسترس خواهند بود.

با توجه به اینکه در این مقاله از SQL Server استفاده خواهد شد، لازم است پکیج SqlServer را نیز نصب کنید تا EF Core بتواند با بکارگیری Functionality مختص SQL، برای تبدیل کدها به Expression Tree و سپس به کوئری قابل اجرا برروی دیتابیس از آن استفاده کند.

در ادامه، برای تعیین Complex Type بودن یک مدل، از ComplexTypeAttribute استفاده خواهد شد که یک Mapping Attribute است و به Attribute هایی گفته می‌شود که در بالای هرکلاس فراخوانی می‌شوند.

نتیجه به شکل زیر خواهد شد:

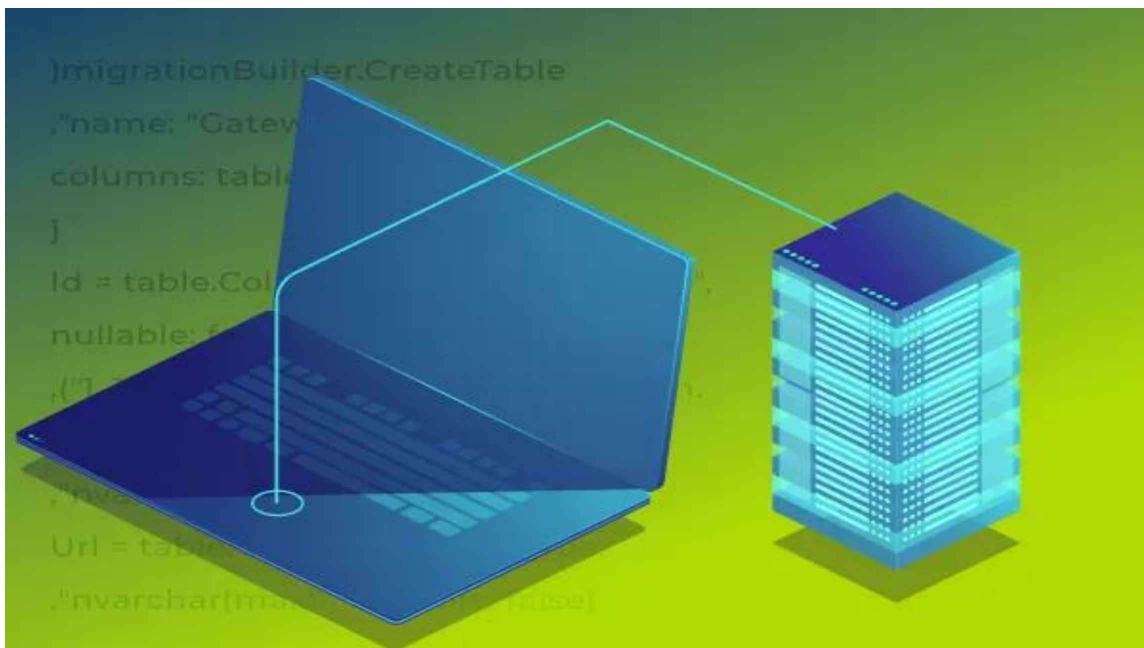
```
[ComplexType]
public class Payment
{
    public readonly decimal Amount;
    public readonly string Currency;
    public readonly decimal Rate;

    public Payment(decimal amount, string currency, decimal rate)
    {
        Amount = amount;
        Currency = currency;
        Rate = rate;
    }
}
```

روش دیگری که برای تعیین ComplexType بودن یک مدل معرفی شد؛ این روش، در متد OnModelCreating در EF Core قابل پیاده‌سازی است.

در مرحله اول، یک کلاس به نام ApplicationDbContext تعریف کنید که از **DbContext** ارث‌بری کند. در داخل آن، متد OnModelCreating را Override کرده و کدهای مربوط به تعریف Complex Type را مشابه زیر تایپ کنید:

```
public class ApplicationDbContext : DbContext
{
    protected override void OnModelCreating(ModelBuilder builder)
    {
        builder.Entity<Gateway>()
            .ComplexProperty(x => x.Payment);
    }
}
```

در متد OnModelCreating، با استفاده از Fluent API مشخص کردیم که Payment یک مدل از نوع Complex Type است. این تکه کد در EF Core تعیین می‌کند که نباید با این Property مانند یک Primitive Type برخورد شود. اگر چنین کانفیگی انجام نشود، امکان بروز خطا یا Inconsistency در زمان نوشتن یا خواندن از دیتابیس وجود خواهد داشت.



تا این گام شرح دادیم که چه مدل‌هایی Complex Type نام دارند و یک مدل برای ذخیره در دیتابیس تعریف کردیم. در قدم بعدی، به‌صورت عملی این مدل را در دیتابیس ذخیره خواهیم کرد.

برای فراهم کردن زیرساخت موردنیاز، **Docker** را برروی سیستم خود نصب کرده و سپس با استفاده از دستور زیر، Sql را pull کنید.

```
docker pull mcr.microsoft.com/mssql/server:2022-latest
```

بعد از دانلود کامل Image دیتابیس، از دستور زیر استفاده کرده تا دیتابیس را Run کنید.

```
docker run -e "ACCEPT_EULA=Y" -e "MSSQL_SA_PASSWORD=Choose@Strong#Password" -p 1433:1433 --name sql1 --hostname sql1 -d mcr.microsoft.com/mssql/server:2022-latest
```

با استفاده از docker run می‌توان image دریافتی را اجرا کرد. با کمک MSSQL_SA_PASSWORD، یک پسورد مناسب برای آن انتخاب کنید. در این آموزش، پسورد انتخاب‌شده برابر با Choose@Strong#Password در نظر گرفته شده است. در صورتی که پسورد انتخاب‌شده شامل حرف، عدد یا کاراکترهای خاص نباشد، احتمالاً دیتابیس پس از اجرا، متوقف می‌شود.

حال قصد داریم به بررسی نحوه ساخت جداول ازطریق EF Core بپردازیم. برای این کار، ابتدا CLI قدرتمند EF Core را ازطریق دستور زیر نصب کنید:

```
dotnet tool install --ignore-failed-sources --add-source
https://api.nuget.org/v3/index.json --global dotnet-ef
```

سپس، با استفاده از دستور زیر، Migration را تولید کنید:

```
dotnet ef migrations add
    --project Infrastructure\Infrastructure.csproj
    --startup-project ComplexTypeInEfCore8\ComplexTypeInEfCore8.csproj
    --context Infrastructure.ApplicationDbContext
    --configuration Debug Initial --output-dir Migrations
```

دستور بالا یک Migration با نام Initial در داخل پوشه Migrations اضافه خواهد کرد. متناسب با ساختار پروژه خود، آن را تغییر دهید. برای راهنمایی بیشتر، می‌توانید به سورس کد پروژه مراجعه کنید.

به موارد زیر توجه کنید:

تگ project– اشاره به پروژه‌ای دارد که در آن کلاس ApplicationDbContext قرار دارد.

تگ startup-project– اشاره به پروژه‌ای دارد که در آن کلاس Program.cs قرار دارد.

فایل Migration تولیدشده به‌صورت زیر خواهد بود:

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.CreateTable(
        name: "Gateway",
        columns: table => new
        {
            Id = table.Column<long>(type: "bigint", nullable: false)
                .Annotation("SqlServer:Identity", "1, 1"),
            Name = table.Column<string>(type: "nvarchar(max)", nullable:
false),
            Url = table.Column<string>(type: "nvarchar(max)", nullable:
false),
            Payment_Amount = table.Column<decimal>(type: "decimal(18,2)",
nullable: false),
            Payment_Currency = table.Column<string>(type: "nvarchar(max)",
nullable: false),
            Payment_Rate = table.Column<decimal>(type: "decimal(18,2)",
nullable: false)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_Gateway", x => x.Id);
        });
}
```

متد Up مربوط به مایگریشن تولیدشده توسط EF Core است و در داخل پوشه Migrations ایجاد می‌شود. در این متد، تابع CreateTable فراخوانی شده که یک جدول به اسم Gateway ایجاد می‌کند.

EF Core به سه طریق، نام جدول را انتخاب خواهد کرد. ابتدا بررسی خواهد کرد که آیا برای کلاس attribute class درج شده است یا خیر. اگر attribute class درج شده باشد، تنظیمات آن را به‌عنوان نام جدول لحاظ می‌کند. در غیر این صورت، نام کلاس به‌عنوان نام جدول انتخاب خواهد شد. روش سوم برای انتخاب نام جدول، کانفیگی است که برای EF Core در کلاس ApplicationDbContext نوشته‌اید. می‌توان در این کلاس مشخص کرد که نام جدول موردنظر برای کلاس Complex Type چه باشد. بنابراین، EF Core به‌صورت خلاصه و به ترتیب، اول کانفیگ‌ها را بررسی می‌کند، سپس به attribute class ها توجه کرده و در آخر به Convention مورد استفاده برای نام‌گذاری کلاس‌ها دقت خواهد کرد. با توجه به اینکه این موارد هیچ یک در این پروژه وجود نداشتند، پس EF Core نام کلاس Gateway را به‌عنوان نام جدول انتخاب کرد.

ستون های Name ,Url و ID متناسب با نام و نوع Property کلاس Gateway ایجاد می‌شوند. به‌عنوان مثال، long در سی شارپ معادل bigint در دیتابیس هست و الی آخر. تا این جا همانند گذشته، دیتابیس و EF Core رفتار متناظر با هر Primitive Type را داشتند. اما EF Core به شیوه دیگری کلاس Payment را درنظر می‌گیرد.

توجه کنید که ستون‌های باقی‌مانده از این جدول حائز اهمیت هستند. EF Core برای هر سه Property کلاس Payment سه پیشوند به‌صورت _Payment در نظر گرفته است؛ یعنی Amount, Currency و Rate هر سه با پیشوند Payment که نام کلاسشان است، در جدول Gateway قرار می‌گیرند و کلاس Payment و Gateway هر دو در یک جدول ذخیره می‌شوند. منتها Property های مربوط به کلاس Payment در ستون‌هایی ذخیره می‌شوند که با پیشوند _Payment مشخص شده‌اند.

در این مثال، بیزینس ما به نحوی بود که می‌بایست یک یا چند Value Object با یک کلاس Entity مورد استفاده قرار می‌گرفتند. به عبارتی، در این جا نیاز داشتیم تا مقادیری که برای Payment و Gateway ذخیره می‌شوند Atomic باشند؛ این یعنی یا Gateway به‌همراه Payment ذخیره می‌شود یا هیچ از مقادیر آن‌ها در دیتابیس قرار نمی‌گیرد. همین موضوع درمورد Fetch کردن داده‌های جدول صدق می‌کند.

مزایای Complex Type در EF Core 8 چیست؟

با استفاده از Complex Type در EF Core 8، می‌توانید کدهای قابل استفاده و به دور از تکرار داشته باشید. به‌عنوان مثال، اگر شما یکسری Property های تکراری را در چندین مدل استفاده کرده‌اید، می‌توانید همه آن‌ها را به‌عنوان یک Complex Type تعریف کرده و سپس آن را در مدل‌های موردنظر خود به کار ببرید. این کار به شما اجازه می‌دهد تا کد Maintainable داشته باشید و همچنین از اشتباهات تکراری جلوگیری کنید. علاوه‌بر این، با استفاده از Complex Type، امکان ایجاد ساختارهای پیچیده‌تری را برای مدل‌ها خواهید داشت و می‌توانید به سادگی مدل‌هایی کاملاً منطبق با دامنه بیزینس خود ایجاد کرده و از بابت شیوه ذخیره و بازیابی آن خیالتان راحت باشد.

جمع بندی: بررسی Complex Type در EF Core 8

در این مقاله به بررسی Complex Type در EF Core 8 پرداختیم و نحوه استفاده از آن را آموزش دادیم. به‌طور کلی، استفاده از نوع Complex Type کمک می‌کند تا بتوانید ساختارهای پیچیده‌تری را برای مدل‌های خود ایجاد کنید. البته باید توجه کنید که در شروع دیزاین پروژه و مدل‌ها، از تحلیل و طراحی اشتباه Complex Type بپرهیزید؛ چراکه ممکن است در آینده نتوانید به سادگی آن را تغییر دهید.