

Entity Framework – Data modelering en Migration Tools

In dit document gaan we verder in op data modellering met EF Core. Het is hier niet de bedoeling om alle mogelijkheden te overlopen: daarvoor verwijzen we naar de online documentatie van Microsoft (<https://docs.microsoft.com/en-us/ef/core/>). Het is wel de bedoeling om via een eenvoudig voorbeeld een aantal van de mogelijkheden te schetsen zodat we aan de slag kunnen gaan.

Entity Types en Properties

Tijdens de eerste kennismaking met EF hebben we gezien hoe we via de code first aanpak een tabel kunnen aanmaken en hoe er automatisch een primaire sleutel (PK) wordt aangemaakt. In dit voorbeeld bekijken we een aantal mogelijkheden om het modelleren verder te specificeren.

Primary Key

De PK kan op verschillende manieren worden gedefinieerd, er is de automatische aanmaak op basis van een naamconventie. Wanneer EF een property vindt met de naam `Id` of `<classnaam>Id` (vb `ReviewId`) zal deze property automatisch de PK worden. Daarnaast kunnen we ook met annotaties werken zoals in het voorbeeld met de klasse `Book`, waar we gekozen hebben om het ISBN-nummer als PK te gaan gebruiken. Het aanduiden van de PK gebeurt dan door de annotatie `[Key]` te plaatsen voor de property.

Voor PKs van het type `int` zal EF impliciet instellen om nieuwe waarden te genereren bij het toevoegen van records. Hierbij zal gebruik gemaakt worden van de mogelijkheden die de databank biedt. In het geval van SQL Server zal dit een `Identity` worden.

Tabellen

Tabellen worden automatisch gemaakt, wanneer er in de Context-klasse een `DbSet<>` property wordt gedefinieerd voor een specifieke klasse. Zo hebben we in de klasse `BookContext` een `DbSet<Book>` en `DbSet<Publisher>` die ervoor zorgt dat beide klassen worden gekoppeld aan een tabel. De `Review` klasse hebben we niet specifiek als `DbSet` opgenomen, maar aangezien deze in de klasse `Book` als navigational property is opgenomen, wordt ook voor deze klasse een tabel aangemaakt in de databank.

Bij de automatische aanmaak van de tabellen wordt de naam van de klasse ook overgenomen als de naam van de tabel. Wens je daarvan af te wijken dan kan je expliciet de tabelnaam opgeven via de annotatie `[Table("tabelnaam")]` zoals we bij de klasse `Publisher` hebben gedaan. Een tweede manier om de tabelnaam op te geven is via de functie `OnModelCreating` in de Context-klasse :

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Book>().ToTable("BookInfo");
}
```

Properties

Ook voor kolommen en properties zijn er mogelijkheden om verder te specificeren, zo kunnen we de naam van een kolom opgeven via de annotatie `[Column("kolomnaam")]` zoals we hebben gedaan bij de property Naam in de klasse Publisher. Ook het type van de kolom kan worden gespecificeerd door middel van annotaties, vb `[Column(TypeName="varchar(500)"]`. Verder zijn er nog mogelijkheden om bijvoorbeeld de maximale lengte op te geven voor het opslaan van een string in de databank `[MaxLength(500)]` en kunnen we ook aangeven of een column wel of niet een NULL-waarde kan bevatten. Dit laatste kan bijvoorbeeld door de annotatie `[Required]` mee te geven, waardoor een kolom geen NULL-waarden kan bevatten. Geven we geen specifieke info mee, dan wordt de volgende keuze automatisch gemaakt : als de property een null-waarde kan hebben (zoals string of int?) dan wordt dit overgenomen in de kolom definitie in het andere geval zal de kolom geen NULL constraint bevatten.

Voorbeeld

De klasse Review, met PK ReviewID (automatisch) en met property Text die in de databank zal beperkt worden tot 500 karakters.

```
public class Review
{
    0 references
    public Review(string text, int stars) ...

    0 references
    public int ReviewID { get; set; }
    [Column(TypeName = "varchar(500)")]
    1 reference
    public string Text { get; set; }
    1 reference
    public int Stars { get; set; }
}
```

De klasse Book heeft een PK ISBN die we aanduiden met de annotatie Key. De kolom Title mag niet NULL zijn en heeft een maximale lengte van 500 in de databank. De variabele Reviews (een navigational property) zorgt ervoor dat er ook voor de klasse een tabel zal worden aangemaakt. Door de variabele NumberOfPages van het type int? Te declareren zal de kolom in de databank wel NULL-waarden mogen bevatten.

```

public class Book
{
    0 references
    public Book(string ISBN, string title, int? numberOfPages) ...

    [Key]
    1 reference
    public string ISBN { get; set; }
    [MaxLength(500)]
    [Required]
    1 reference
    public string Title { get; set; }
    1 reference
    public int? NumberOfPages { get; set; }
    0 references
    public List<Review> Reviews { get; private set; } = new List<Review>();
}

```

Voor de klasse Publisher hebben we gekozen om de tabelnaam 'PublisherInfo' te noemen en de property Name zal als kolomnaam 'PublisherName' krijgen. De PK is het Id-veld dat op basis van de naming convention zal worden toegekend.

```

[Table("PublisherInfo")]
2 references
public class Publisher
{
    0 references
    public Publisher(string name) ...

    0 references
    public int Id { get; set; }
    [Column("PublisherName")]
    1 reference
    public string Name { get; set; }
}

```

Onze Context-klasse bevat een DbSet voor Books en Publishers die ervoor zorgt dat beide tabellen worden aangemaakt. Bij de OnModelCreating methode maken we gebruik van de modelBuilder om de naam van de tabel voor de klasse Book in te stellen op 'BookInfo'.

Voor de configuratie van de databank maken we gebruik van de optionsBuilder.

```

public class BookContext : DbContext
{
    0 references
    public DbSet<Book> Books { get; set; }
    0 references
    public DbSet<Publisher> Publishers {get;set;}
    0 references
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Book>().ToTable("BookInfo");
    }
    0 references
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(@"Data Source=lenovo-pc\squlexpress;Initial Catalog=bookDB;Integrated
    }
}

```

Stoppen we deze klassen in het project EFmodelling (.NET Core console applicatie) en installeren we de nodige NuGet packages dan kunnen we met de migration tools de databank initialiseren.

```

PM> Add-Migration InitialCreateBook -project EFmodelling
Build started...
Build succeeded.
To undo this action, use Remove-Migration.

```

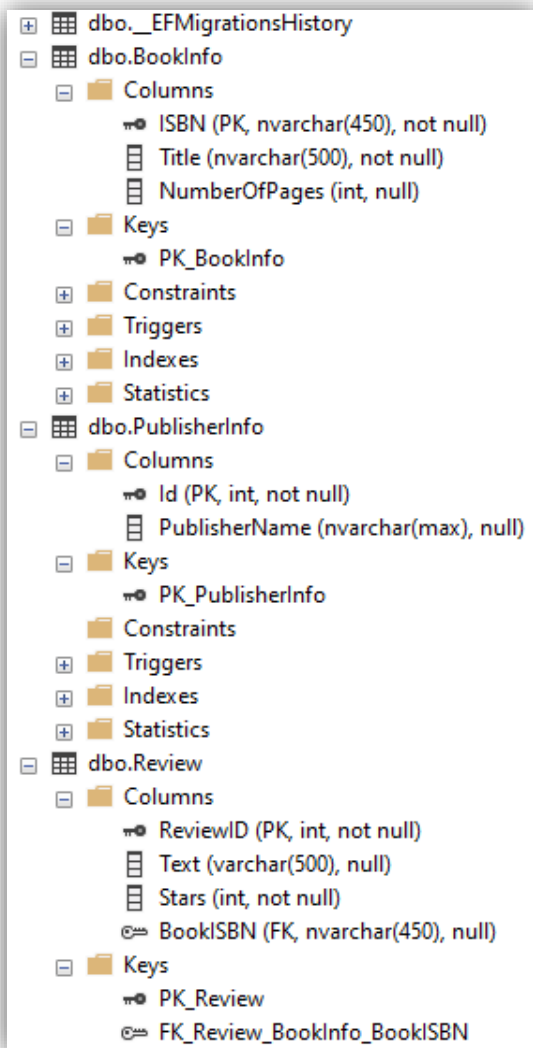
```

PM> Update-Database -project EFmodelling
Build started...
Build succeeded.
Applying migration '20200327084330_InitialCreateBook'.
Done.

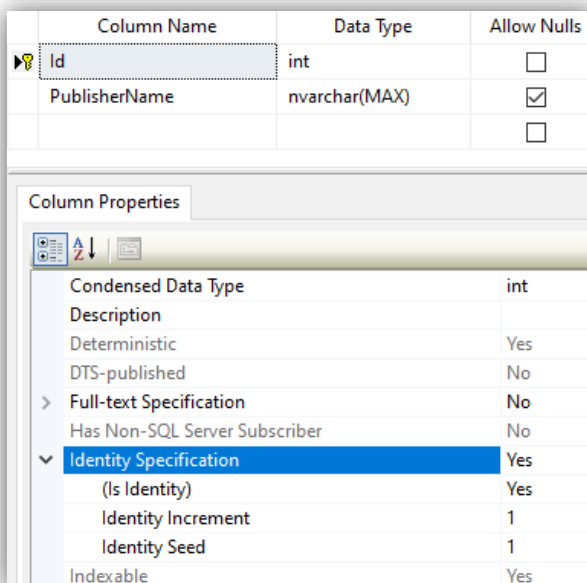
```

Opmerking : het project EFmodelling maakt deel uit van een Solution die meerdere projecten omvat en daarom moet bij het gebruik van de tools ook aangegeven worden welk project er moet worden gebruikt.

Het resultaat ziet er dan als volgt uit :



Bekijken we de tabel PublisherInfo in detail (design view) dan zien we dat de PK ingesteld is als Identity.



Shadow properties en Indexen

We passen de klasse Book nu aan zodanig dat deze nu ook beschikt over een Publisher property. Er zijn nu twee mogelijkheden voor de relatie te implementeren : ofwel voeren we enkel de property Publisher toe en dan zal EF een shadow property aanmaken voor de Foreign Key (FK) ofwel voorzien we zelf een extra veld (sleutel) waarmee we naar dit gelinkt object verwijzen. In ons voorbeeld kiezen we voor optie 2 en voegen we de property PublisherId toe samen met de property Publisher. Voor meer info over shadow properties verwijzen we naar <https://www.learnentityframeworkcore.com/model/shadow-properties> en de documentatie van Microsoft zelf <https://docs.microsoft.com/en-us/ef/core/modeling/shadow-properties> .

De aangepaste klasse Book ziet er dan als volgt uit :

```
public class Book
{
    0 references
    public Book(string ISBN, string title, int? numberOfPages) ...

    [Key]
    1 reference
    public string ISBN { get; set; }
    [MaxLength(500)]
    [Required]
    1 reference
    public string Title { get; set; }
    1 reference
    public int? NumberOfPages { get; set; }
    0 references
    public List<Review> Reviews { get; private set; } = new List<Review>();
    1 reference
    public int PublisherId { get; set; }
    0 references
    public Publisher Publisher { get; set; }
}
```

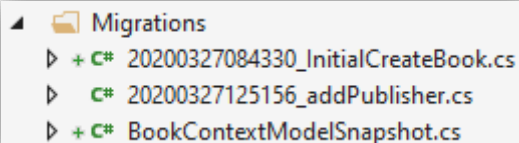
EF laat ook toe om indexen te definiëren voor bepaalde kolommen. Dit kan door gebruik te maken van de modelBuilder methode HasIndex, waarbij door middel van een lambda functie de index wordt bepaald. We kiezen er nu voor om een index te maken voor de kolom PublisherId in de tabel Books. Doordat we expliciet de property PublisherId hebben toegevoegd in de klasse Book is dit nu eenvoudig te realiseren.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Book>().ToTable("BookInfo");
    modelBuilder.Entity<Book>().HasIndex(b => b.PublisherId);
}
```

Het doorvoeren van deze aanpassingen kan door middel van de Migration tools. We voegen een nieuwe migratie in via de package manager console als volgt, met als naam addPublisher :

```
PM> Add-Migration addPublisher -project EFmodelling
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
```

Er verschijnt in de migrations folder van ons project nu een nieuwe klasse die de naam van onze migratie bevat.



```

Migrations
├── + C# 20200327084330_InitialCreateBook.cs
├── C# 20200327125156_addPublisher.cs
├── + C# BookContextModelSnapshot.cs
```

Deze klasse bevat de aanpassingen aan het model die we hebben doorgevoerd.

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.AddColumn<int>(
        name: "PublisherId",
        table: "BookInfo",
        nullable: false,
        defaultValue: 0);

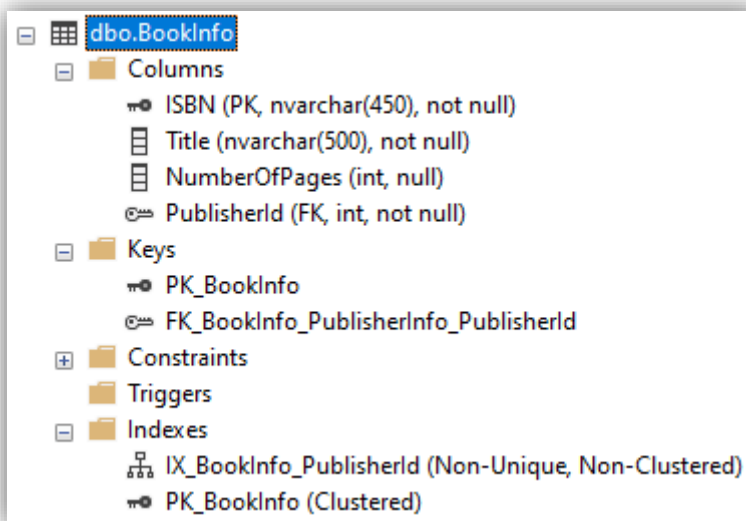
    migrationBuilder.CreateIndex(
        name: "IX_BookInfo_PublisherId",
        table: "BookInfo",
        column: "PublisherId");

    migrationBuilder.AddForeignKey(
        name: "FK_BookInfo_PublisherInfo_PublisherId",
        table: "BookInfo",
        column: "PublisherId",
        principalTable: "PublisherInfo",
        principalColumn: "Id",
        onDelete: ReferentialAction.Cascade);
}
```

Het effectief doorvoeren in de databank doen we weer met het commando Update-Database.

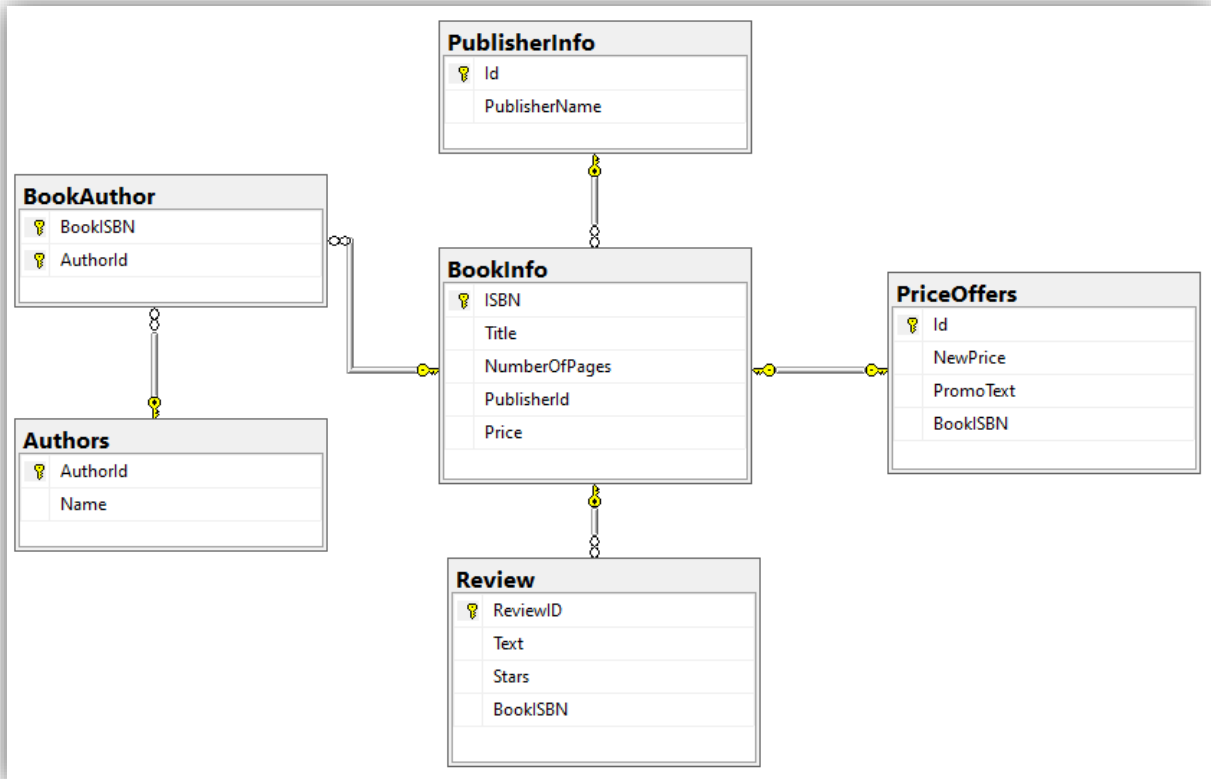
```
PM> Update-Database -project EFmodelling
Build started...
Build succeeded.
Applying migration '20200327125156_addPublisher'.
Done.
```

Bekijken we het resultaat dan constateren we dat een extra veld PublisherId is aangemaakt, er een FK is toegevoegd en de gevraagde index is aangemaakt.



Relaties

Een belangrijk aspect bij het modelleren zijn de relaties tussen verschillende klassen. In vorige paragrafen zijn relaties al even aan bod gekomen, maar we gaan er hier nog wat verder op in. We voegen een 1-op-1 relatie toe tussen Book en PriceOffer waarbij er voor één specifiek boek een promo kan worden gegeven. Daarnaast voegen we ook een veel-op-veel relatie toe tussen Author en Book, een boek kan door verschillende auteurs worden geschreven, maar een auteur kan ook verschillende boeken schrijven natuurlijk. In databank termen wordt dit door middel van een tussentabel gerealiseerd. Een overzicht van het databankschema vind je in de volgende figuur.



1-op-1 relatie

We starten met de 1-op-1 relatie tussen PriceOffer en Book. De klasse PriceOffer kunnen we als volgt aanmaken, met een referentie naar Book en de daarbij horende BookISBN sleutel.

```

public class PriceOffer
{
    0 references
    public int Id { get; set; }
    0 references
    public double NewPrice { get; set; }
    0 references
    public string PromoText { get; set; }
    1 reference
    public string BookISBN { get; set; }
    1 reference
    public Book Book { get; set; }
}
  
```

In de klasse Book voorzien we dan het extra veld PriceOffer.

```

public PriceOffer PriceOffer { get; set; }
  
```

Om nu aan te geven op welke manier de relatie moet worden opgebouwd, maken we gebruik van de modelBuilder. Daarin maken we gebruik van de HasOne-methode in combinatie met WithOne. Belangrijk bij een 1-op-1 relatie is de Foreign Key eigenschap en daarbij het aanduiden van de hoofdentiteit en de afhankelijke entiteit. In dit voorbeeld is de hoofdentiteit Book en de afhankelijke entiteit PriceOffer. Om dit nu te modelleren maken we gebruik van het volgende statement.

```
modelBuilder.Entity<Book>()  
    .HasOne(b => b.PriceOffer)  
    .WithOne(i => i.Book)  
    .HasForeignKey<PriceOffer>(b => b.BookISBN);
```

Voor de entiteit Book geven we aan dat deze een link heeft met PriceOffer door middel van de lambda-expressie in de HasOne-methode. Voor de omgekeerde relatie gebruiken we eveneens een lambda-expressie die de link aangeeft naar Book. Verder definiëren we nog de FK bij de afhankelijke entiteit door middel van deHasForeignKey-methode waarbij de sleutel wordt gedefinieerd door middel van een lambda-expressie.

(<https://www.learnentityframeworkcore.com/configuration/one-to-one-relationship-configuration>)

Veel-op-veel relatie

Deze relatie is in EF Core nog niet beschikbaar in een 'propere' vorm en moet gemodelleerd worden door middel van expliciet gebruik te maken van een klasse die de koppeltabel voorstelt. De koppeltabel bevat de IDs voor zowel Book als Author en kunnen we door middel van de volgende klasse modelleren :

```
public class BookAuthor  
{  
    1 reference  
    public string BookId {get;set;}  
    0 references  
    public Book Book { get; set; }  
    1 reference  
    public int AuthorId { get; set; }  
    0 references  
    public Author Author { get; set; }  
}
```

In tegenstelling tot onze vorige klassen bestaat de PK voor deze tabel uit twee velden en moeten we dit ook expliciet aangeven. Dit kan door middel van het volgende statement toe te voegen in de OnModelCreating methode :

```
modelBuilder.Entity<BookAuthor>().HasKey(x => new { x.BookISBN, x.AuthorId });
```

In de Author klasse leggen we de link naar de boeken door middel van een property die een collectie van BookAuthor klassen voorstelt.

```
public class Author
{
    0 references
    public int AuthorId { get; set; }
    0 references
    public string Name { get; set; }
    0 references
    public ICollection<BookAuthor> BooksLink { get; set; }
}
```

Ook in de klasse Book voorzien we een property die een collectie naar deze koppelklasse bevat.

```
public class Book
{
    0 references
    public Book(string ISBN, string title, int? numberOfPages) ...

    [Key]
    1 reference
    public string ISBN { get; set; }
    [MaxLength(500)]
    [Required]
    1 reference
    public string Title { get; set; }
    1 reference
    public int? NumberOfPages { get; set; }
    0 references
    public List<Review> Reviews { get; private set; } = new List<Review>();
    1 reference
    public int PublisherId { get; set; }
    0 references
    public Publisher Publisher { get; set; }
    0 references
    public double Price { get; set; }
    1 reference
    public PriceOffer PriceOffer { get; set; }
    0 references
    public ICollection<BookAuthor> AuthorsLink { get; set; }
}
```

Implementatie

We hebben nu de twee extra relaties gemodelleerd in onze klassen, wat ons nu nog te doen staat is de BookContext-klasse aanvullen met de DbSet's voor Author en PriceOffer zodat beiden beschikbaar zijn.

```

public class BookContext : DbContext
{
    0 references
    public DbSet<Book> Books { get; set; }
    0 references
    public DbSet<Publisher> Publishers {get;set;}
    0 references
    public DbSet<PriceOffer> PriceOffers { get; set; }
    0 references
    public DbSet<Author> Authors { get; set; }
    0 references
    protected override void OnModelCreating(ModelBuilder modelBuilder) [...]
    0 references
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) [...]
}

```

De OnModelCreating-klasse ziet er dan als volgt uit :

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Book>().ToTable("BookInfo");
    modelBuilder.Entity<Book>().HasIndex(b => b.PublisherId);
    modelBuilder.Entity<BookAuthor>().HasKey(x => new { x.BookId, x.AuthorId });
    modelBuilder.Entity<Book>()
        .HasOne(b => b.PriceOffer)
        .WithOne(i => i.Book)
        .HasForeignKey<PriceOffer>(b => b.BookISBN);
}

```

Voor het uitvoeren van de aanpassingen in ons model kunnen we weer gebruik maken van de Migration tools.

```

PM> Add-Migration addRelations -project EFmodelling
Build started...
Build succeeded.
To undo this action, use Remove-Migration.

```

Een extra klasse verschijnt weer in de Migrations folder in Visual Studio.

```

Migrations
└─ 20200327084330_InitialCreateBook.cs
└─ 20200327125156_addPublisher.cs
└─ 20200329151304_addRelations.cs
└─ ✓ 20200329151304_BookContextModelSnapshot.cs

```

De aanpassingen doorvoeren in de databank kan via het Update-Database commando.

```

PM> Update-Database -project EFmodelling
Build started...
Build succeeded.
Applying migration '20200327171621_addRelations'.
Done.

```

Als resultaat krijgen we nu, een extra record in de _EFMigrationsHistory tabel met de naam van de nieuwe migratie.

MigrationId	ProductVersion
20200327084330_InitialCreateBook	3.1.3
20200327125156_addPublisher	3.1.3
20200327171621_addRelations	3.1.3

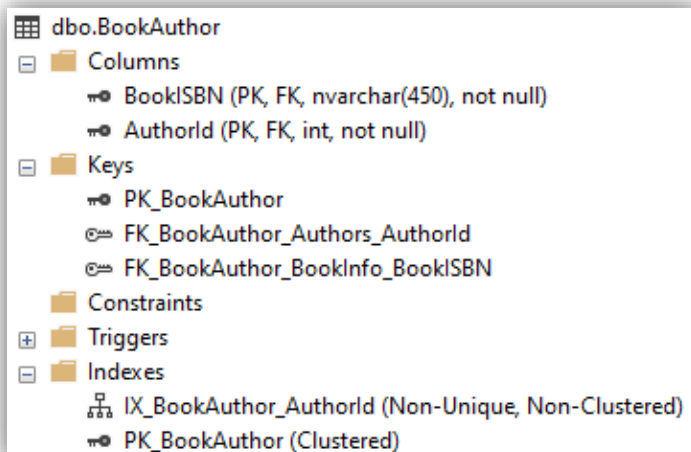
De nieuwe tabel PriceOffers is ook toegevoegd, met daarin de FK naar de BooksInfo tabel via het veld BookISBN. Er wordt ook automatisch een index aangemaakt voor elke FK.

dbo.PriceOffers
Columns
Id (PK, int, not null)
NewPrice (float, not null)
PromoText (nvarchar(max), null)
BookISBN (FK, nvarchar(450), null)
Keys
PK_PriceOffers
FK_PriceOffers_BookInfo_BookISBN
Constraints
Triggers
Indexes
IX_PriceOffers_BookISBN (Unique, Non-Clustered, Filtered)
PK_PriceOffers (Clustered)

Er worden ook twee nieuwe tabellen aangemaakt voor Authors en BookAuthors (de koppeltabel).

dbo.Authors
Columns
AuthorId (PK, int, not null)
Name (nvarchar(max), null)
Keys
PK_Authors
Constraints
Triggers
Indexes
PK_Authors (Clustered)

In de BookAuthor tabel is ook te zien dat de PK bestaat uit zowel de kolom BookISBN als de kolom AuthorId. Voor beiden is er ook een FK en index aangemaakt. Je ziet geen specifieke index voor BookISBN aangezien deze door middel van de PK-index wordt gerealiseerd.



In de BookInfo tabel zien we enkel de extra kolom Price die we hebben toegevoegd.

