# Entity Framework (EF) Core – Version 1.1

EF Core was rewritten from scratch. Where EF Core can be used? **Table 1** shows the answer.

| | .Net 4.5.1+ | .NET Core | UWP |
|---|---|---|---|
| OS | Windows | Windows, macOS, Linux | Windows 10 |
| App Types | Any .NET 4.5.1+ app | ASP.NET Core (Web apps & APIs)<br>Libraries<br>Services<br>Console Apps | Mobile<br>PC<br>HoloLens<br>Xbox<br>Surface Hub<br>IoT Devices |

**Table 1.** Where EF can be used

## Features

EF Core is like EF 6, as Figure 1 below shows some of EF 6 features are present, other never will be.

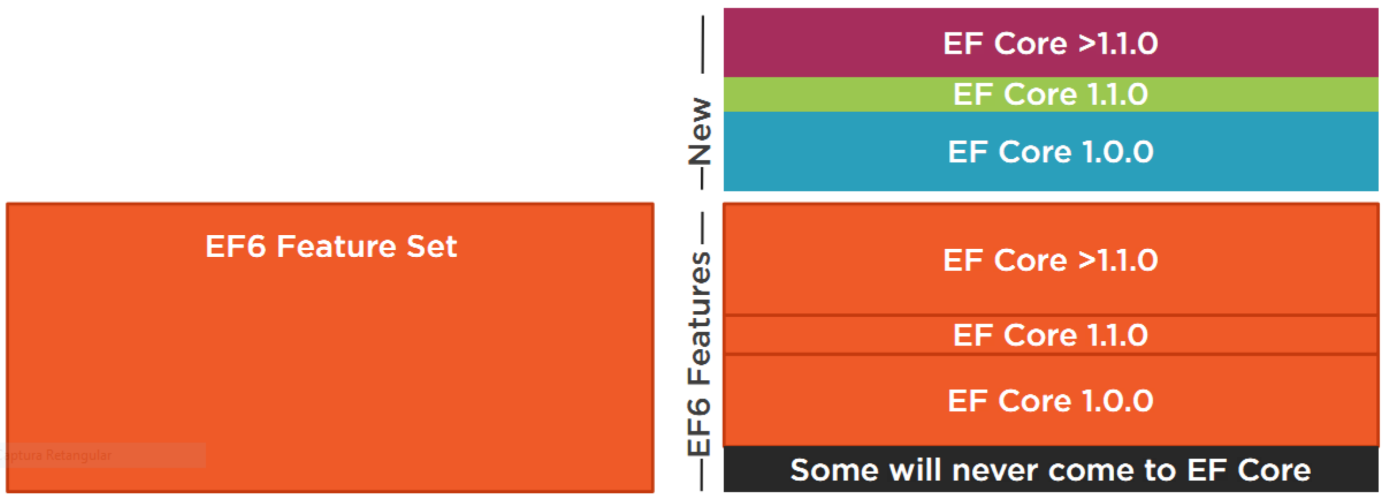


**Figure 1**. Features of EF

## Features not coming

### EDX/Designer in EF Core

There is no designer for EF Core to Database First strategy. Alternatives are in third party solutions like Devart Entity Developer Designer and LLBLGen Pro.

Despite there is no official designer from Microsoft, EF Core offers the EF Core Migration commands, specially the scaffold command which discovers the logic in a database and creates a class model based on that.

## Familiar features

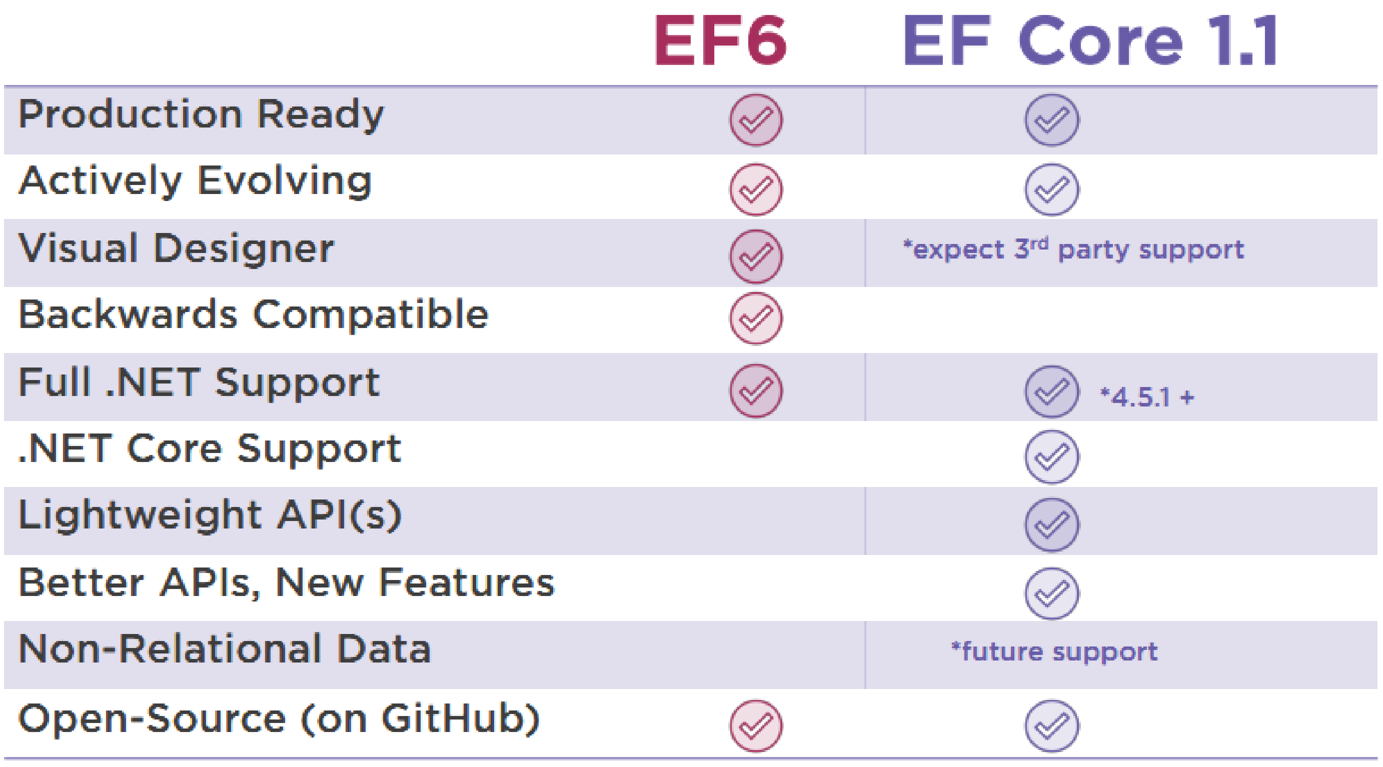
EF Core is still Entity Framework so the following features are present: Data Models, DbContext API, Migrations, LINQ to Entities, Change Tracking, SaveChanges.

## New features

- Batch Insert, Update and Delete operations
- Unique constraints can be defined
- Smarter queries, the engine now can determine what part of a query can be converted to SQL or executed in memory. This allows the use of inline functions in queries
- Smarter disconnected patterns where implemented
- In-Memory provider for testing, tests may not hit databases
- Extensible & IoC friendly, parts of EF Core can be easily replaced using injection (for instance logging)
- Mapping to backing fields and IEnumerables
- Smarter and simpler mappings

## What about EF 6?

EF 6 enters now in a ***maintenance stage***. No new features will be implemented. The Figure 2 compares EF6 and EF Core.



| | EF6 | EF Core 1.1 |
|---|---|---|
| Production Ready | ✓ | ✓ |
| Actively Evolving | ✓ | ✓ |
| Visual Designer | ✓ | *expect 3rd party support |
| Backwards Compatible | ✓ | |
| Full .NET Support | ✓ | ✓ *4.5.1 + |
| .NET Core Support | | ✓ |
| Lightweight API(s) | | ✓ |
| Better APIs, New Features | | ✓ |
| Non-Relational Data | | *future support |
| Open-Source (on GitHub) | ✓ | ✓ |

**Figure 2**. Comparing EF 6 and EF Core

## Exploring EF Core, the basics

### Setting up a solution

In VS2017 create a new empty solution, EFCoreReview targeting .net 4.6.X. Now add a new Class Library (.Net Standard), targeting also .net 4.6.X, named EFCoreReview.Domain. In Domain we create 2 classes: Book and Chapter (see download links). A second class library project is added to the solution, named EFCoreReview.Data. In this project we add the Microsoft.EntityFrameworkCore.SqlServer package, right clicking on the solution and picking Manage Nuget Packages.

We declare a new class in this project, named BookContext as seen in List 1.

```
    public class BookContext: DbContext
    {
        public DbSet<Book> Books { get; set; }
        public DbSet<Chapter> Chapters { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer(
                "Server = (localdb)\\mssqllocaldb; Database= BookDataCore; Trusted_Connection =
True;");
        }
    }
```

**List 1.** DbContext

EF 6 can infer provider if missing, infer connection string if missing, create database if missing, use many IDatabaseInitializers (DropCreateDatabaseIfModelChanges, DropCreateDatabaseAlways…). All this **is gone** in EF Core. We must specify database provider and connection in EF Core. This is done in DbContext class overriding the new method OnConfiguring. Details can be stored in .net app/web.config.

## Creating the Database

### Migrations

Using the Model First approach we can create the database based in our model. For that we use Migrations and its basic workflow is: create/change model -> create a migration file -> apply migration to DB or script.

As mentioned before, EF.Core is modular now. Migrations are part of a new package: Microsoft.EntityFrameworkCore.Tools. We add this package to the EFCoreReview.Data project.

Once we have migrations installed, we've the following commands:

| | |
|---|---|
| `Add-Migration` | Adds a new migration. |
| `Drop-Database` | Drops the database. |
| `Remove-Migration` | Removes the last migration. |
| `Scaffold-DbContext` | Scaffolds a DbContext and entity types for a database. |
| `Script-Migration` | Generates a SQL script from migrations. |
| `Update-Database` | Updates the database to a specified migration. |

We start running the add-migration init command in the Package Manager Console (EFCoreReview.Data must be the startup project). With that we get a new file in the project (Figure 3)

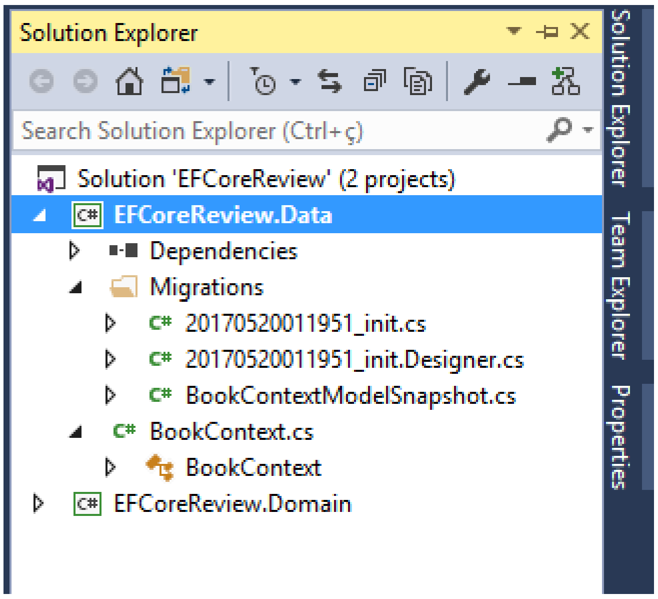

Solution 'EFCoreReview' (2 projects)
EFCoreReview.Data
Dependencies
Migrations
20170520011951_init.cs
20170520011951_init.Designer.cs
BookContextModelSnapshot.cs
BookContext.cs
BookContext
EFCoreReview.Domain

**Figure 3**. Solution Explorer

The file BookConextModelSnapshot.cs is where EF.Core Migrations keeps track of the current state of the model. So we can say that now Migrations are now source-control friendly. In EF 6 and before, the snapshots were stored in the database, being difficult to include them in source-control. Now it is a file, is part of project and is easy to be in the source-control.

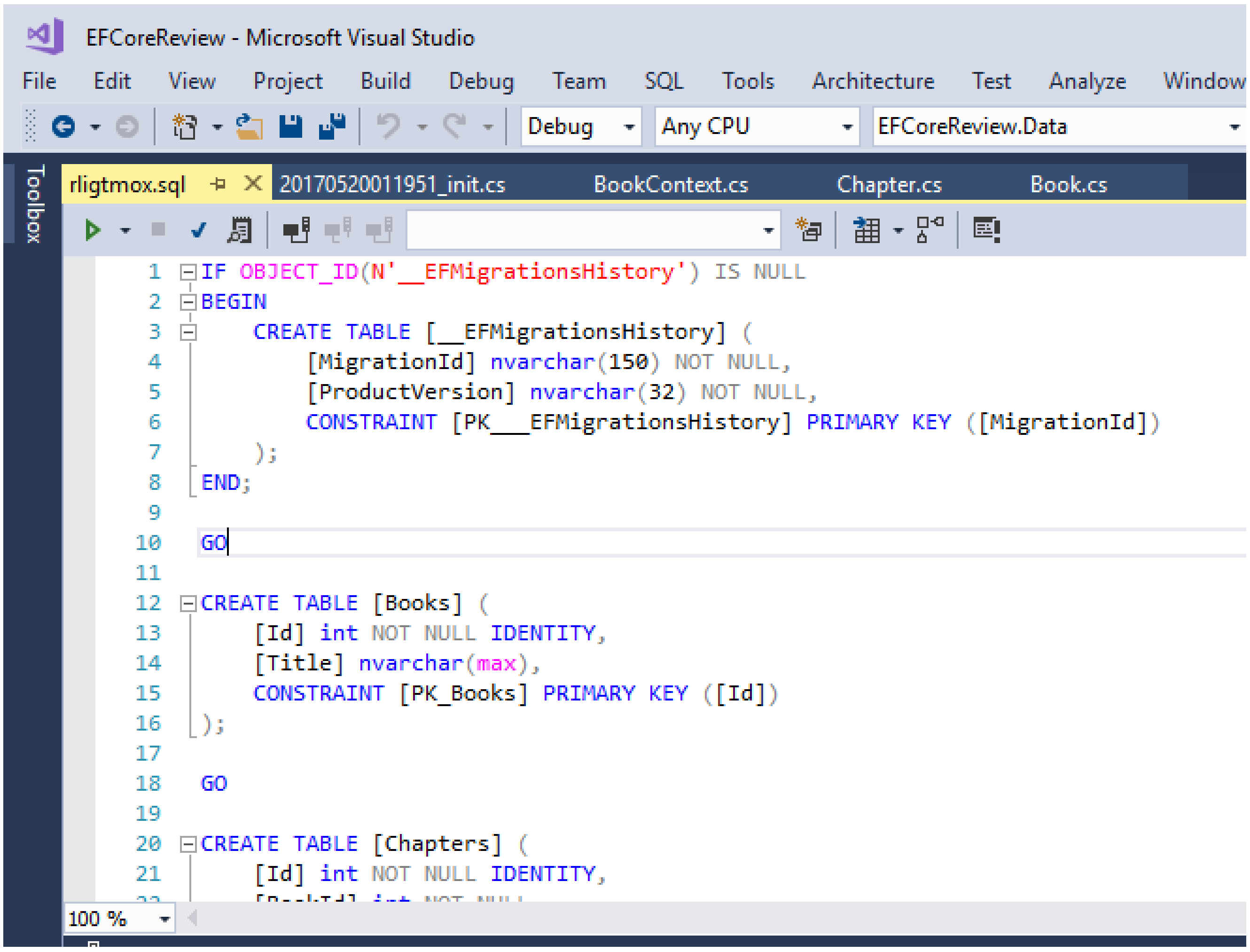
If you take a look at 20170520011951_init.cs file you will note EF.Core Migrations read our BookContext class and identified the configuration, it's detected we are using SQL Server and configured migrations for that.

Before creating the DB let's script it. In EF 6 this was possible using the Update-Database command. It has a parameter for scripting the DB. This wasn't intuitive. Now with EF.Core we have a separate command: Script-Migration. The Script-Migration command behavior also changed in EF.Core (Table 2)

| Version | Command | Behavior |
|---|---|---|
| EF4 – EF6 | Update-Database-Script | Scripts the latest migration (default) |
| EF Core | Script-Migration | Scripts all migrations (default |
| | Script-Migration -Idempotent | Scripts all migrations with If/Then logic |
| | Script-Migration –From [migration] -To [migration] | Controls which migrations are scripted |

**Table 2.** Migration behavior

As this is our first run, we run Script-Migration command in the Package Manager Console and we get the T-SQL according to the migrations (Figure 4).


EFCoreReview - Microsoft Visual Studio
File Edit View Project Build Debug Team SQL Tools Architecture Test Analyze Window
Debug
Any CPU
EFCoreReview.Data
Toolbox
rligtmox.sql
20170520011951_init.cs
BookContext.cs
Chapter.cs
Book.cs
100 %

```sql
IF OBJECT_ID(N'__EFMigrationsHistory') IS NULL
BEGIN
    CREATE TABLE [__EFMigrationsHistory] (
        [MigrationId] nvarchar(150) NOT NULL,
        [ProductVersion] nvarchar(32) NOT NULL,
        CONSTRAINT [PK___EFMigrationsHistory] PRIMARY KEY ([MigrationId])
    );
END;

GO

CREATE TABLE [Books] (
    [Id] int NOT NULL IDENTITY,
    [Title] nvarchar(max),
    CONSTRAINT [PK_Books] PRIMARY KEY ([Id])
);

GO

CREATE TABLE [Chapters] (
    [Id] int NOT NULL IDENTITY,
```

**Figure 4**. T-SQL

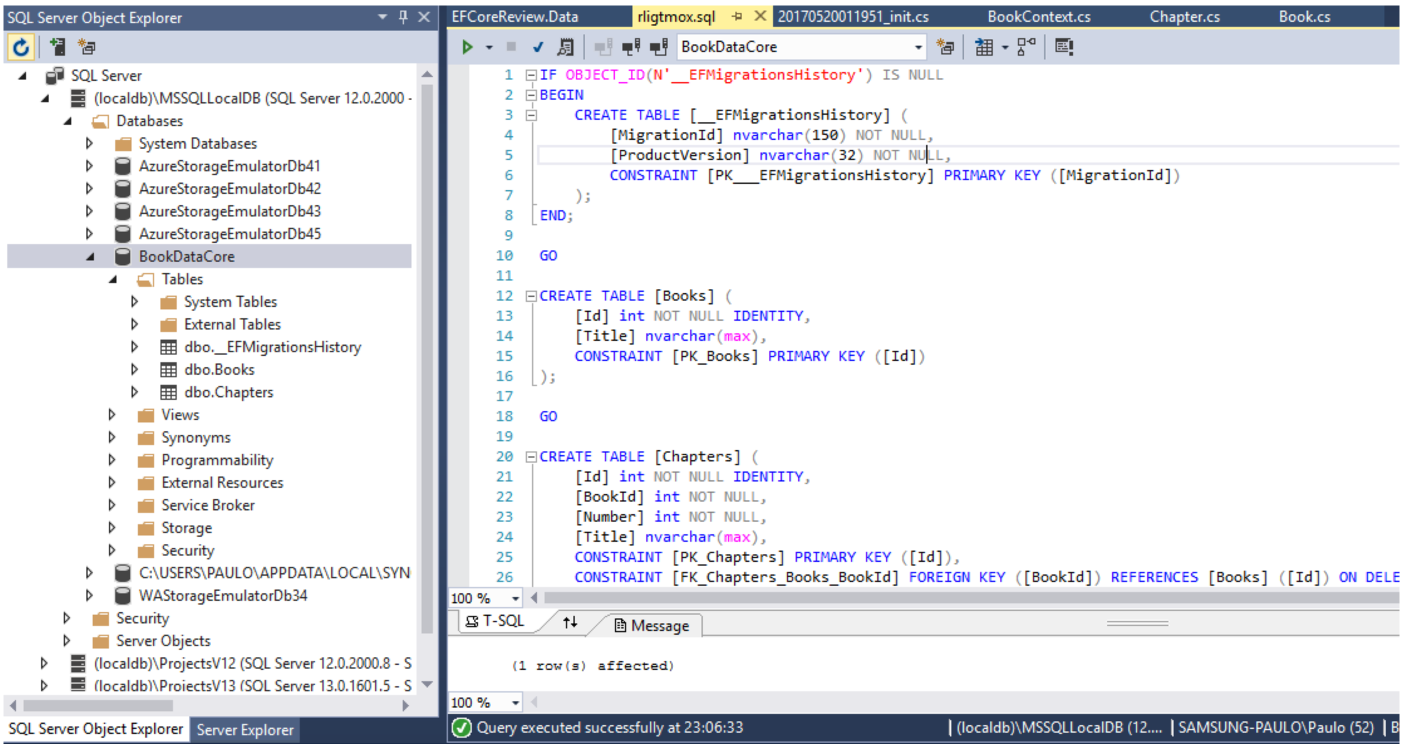
**When use Script-Migration and Update-Database?**

- Script-Migration: offers more control on when and how apply it to DB, focus on production databases, allows needed tweaks to be applied by a DB expert.
- Update-Database: best for local database development.

**Creating direct from migrations**

If we use decide for using the generated script to create the database, we must create it first. But if we use the update-database command, EF.Core checks if DB exists and creates it if necessary.

For this review, I will create the database and execute the generated script (Figure 5).



```sql
IF OBJECT_ID(N'__EFMigrationsHistory') IS NULL
BEGIN
    CREATE TABLE [__EFMigrationsHistory] (
        [MigrationId] nvarchar(150) NOT NULL,
        [ProductVersion] nvarchar(32) NOT NULL,
        CONSTRAINT [PK___EFMigrationsHistory] PRIMARY KEY ([MigrationId])
    );
END;

GO

CREATE TABLE [Books] (
    [Id] int NOT NULL IDENTITY,
    [Title] nvarchar(max),
    CONSTRAINT [PK_Books] PRIMARY KEY ([Id])
);

GO

CREATE TABLE [Chapters] (
    [Id] int NOT NULL IDENTITY,
    [BookId] int NOT NULL,
    [Number] int NOT NULL,
    [Title] nvarchar(max),
    CONSTRAINT [PK_Chapters] PRIMARY KEY ([Id]),
    CONSTRAINT [FK_Chapters_Books_BookId] FOREIGN KEY ([BookId]) REFERENCES [Books] ([Id]) ON DELE
```

```
(1 row(s) affected)
```

```
Query executed successfully at 23:06:33 | (localdb)\MSSQLLocalDB (12.... | SAMSUNG-PAULO\Paulo (52) | B
```

**Figure 5**. Database ready

**Many-To-Many Relations**

EF 6 can infer the database join table, performing some “magic” at runtime. Also, we can define literally a domain entity representing the join table.

In EF.Core the only way is to **create a domain entity representing the join**, so we control relationship in code.

Our domain is related to books, and books have authors. Let’s create this relation. In our domain we add a new class, Author (Id, Name) and another class representing the join between authors and book. See the List 2.

```csharp
public class AuthorBook
{
    public int AuthorId { get; set; }
    public Author Author { get; set; }
    public int BookId { get; set; }
    public Book Book { get; set; }
}
```

**List 2**. AuthorBook class

This ‘join class’ has only the ids and navigation properties. If it was required we could have more properties defining this relationship, for example an Order property that represents who is the first, second, third author.

Now in the Author class where can add a List of AuthorBook, so we can know which books the author wrote and in Book class we have the same (List 3).

```csharp
public class Author
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<AuthorBook> Books { get; set; }

    public Author()
    {
        Books = new List<AuthorBook>();
    }
}
```

```
    public class Book
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public List<Chapter> Chapters { get; set; }
        public List<AuthorBook> Authors { get; set; }

        public Book()
        {
            Chapters = new List<Chapter>();
            Authors = new List<AuthorBook>();
        }
    }
```

**List 3.** Navigation properties

Following the relational model, our join table primary key should be composed by the primary keys of the tables. We expressed that in our model, see the AuthorBook class, it has the primary keys from the tables involved. To express that in our data model we must make the mapping of that.

In BookContext class we override the method OnModelCreating, as follows (List 4).

```
        protected override void OnModelCreating(ModelBuilder modelbuilder)
        {
            modelbuilder.Entity<AuthorBook>()
                .HasKey(s => new { s.AuthorId, s.BookId });
            base.OnModelCreating(modelbuilder);
        }
```

**List 4.** Mapping

And we run the Add-Migration JoinTable command (‘JoinTable’ parameter is the name for the migration).

**One-to-One Relations**

Thanks to support of unique constraints and foreign constraints, EF Core is smarter. The rules for a one-to-one mapping are seen in Table 3.

| EF 6 | EF Core |
|---|---|
| Requires navigation properties on both ends | Requires navigation properties on both ends |
| Primary key of dependent is also FK to principal | FK property used to infer principal and dependent |
| Mark dependent navigation as required, otherwise 1:0..1<br>Or Fluent API to specify principal and dependent | Mark dependent navigation as required, otherwise 1:0..1<br>Or simpler API: HasOne, WithOne |
| | |

**Table 3**. One to one mapping

Let’s change our domain. We know books might have a summary, so we create a new class Summary and reference it in Book and vice versa (List 5).

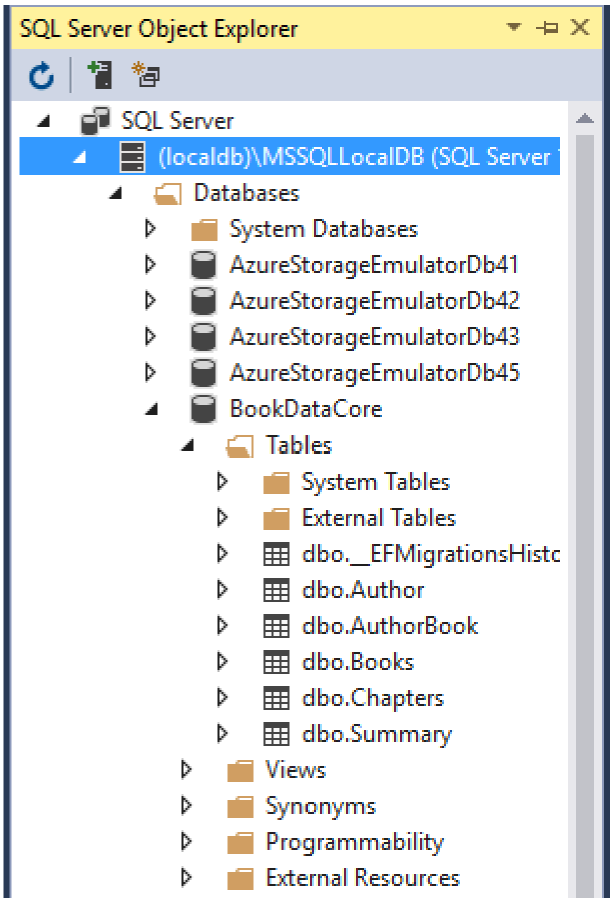
```
    public class Summary
    {
        public int Id { get; set; }
        public string Description { get; set; }
        public int BookId { get; set; }
        public Book Book { get; set; }
    }


    public class Book
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public List<Chapter> Chapters { get; set; }
        public List<AuthorBook> Authors { get; set; }

        public Summary Summary { get; set; }

        public Book()
        {
            Chapters = new List<Chapter>();
            Authors = new List<AuthorBook>();
        }
    }
```

**List 5**. New classes

Now we run the Add-Migration OneToOne command. See the new migration file created.

To script our changes, we can use the script-migration command, but remember, this will script all migrations. We can use also the script-migration -idempotent. In this case all migrations are scripted, but a logic is built within, checking preexistence of objects. Or, we can use script-migration from JoinTable, where JoinTable is the migration name create before. If we only use the parameter 'from', the script will start from JoinTable to the last migration.

For this review, I will use the script-migration -idempotent command and apply the generated scripty manually and we can see the new tables (Figure 6).


SQL Server Object Explorer

```
◢ SQL Server
  ◢ (localdb)\MSSQLLocalDB (SQL Server
    ◢ Databases
      ▷ System Databases
      ▷ AzureStorageEmulatorDb41
      ▷ AzureStorageEmulatorDb42
      ▷ AzureStorageEmulatorDb43
      ▷ AzureStorageEmulatorDb45
      ◢ BookDataCore
        ◢ Tables
          ▷ System Tables
          ▷ External Tables
          ▷ dbo.__EFMigrationsHistc
          ▷ dbo.Author
          ▷ dbo.AuthorBook
          ▷ dbo.Books
          ▷ dbo.Chapters
          ▷ dbo.Summary
        ▷ Views
        ▷ Synonyms
        ▷ Programmability
        ▷ External Resources
```

**Figure 6**. Database

## Insert, Update, and Delete simple objects

We add a new project to the solution, a Console application targeting .NET 4.6.2 and add Microsoft.EntityFrameworkCore.SqlServer packager. With this application, we simulate CRUD operations for simple objects.

The method InsertBook() shows how to insert an object (List 6).

```
        private static void InsertBook()
        {
            var book = new Book() { Title = "C# 7 and .NET Core: Moder Cross-Platform Development"
};
            using (var context = new BookContext())
            {
                context.Books.Add(book);
                context.SaveChanges();
            }
        }
```

**List 6**. InsertBook method

Even EF Core is a new implementation and has differences when comparing to EF 6, the pattern is the same here. Before running this method let's include a logger, EF Core has news about that.

EF 6 has a default System.Data.Entity.Internal.InternalContext.Log class, in EF Core we have the new interface *Microsoft.Extensions.Logging.ILoggerProvider*. This is one proof of the vision for EF, now it is extensible. Implementing that interface allow us to write the logging the way we need.

For the simplicity, we grab an implementation from https://docs.microsoft.com/en-us/ef/core/miscellaneous/logging into EFReview.Data project.

The only change made is the Log method, we remove line that saves to a file, leaving only a visual log to Console (List 7).

```
            public void Log<TState>(LogLevel logLevel, EventId eventId, TState state, Exception
exception, Func<TState, Exception, string> formatter)
            {
                Console.WriteLine(formatter(state, exception));
            }
```

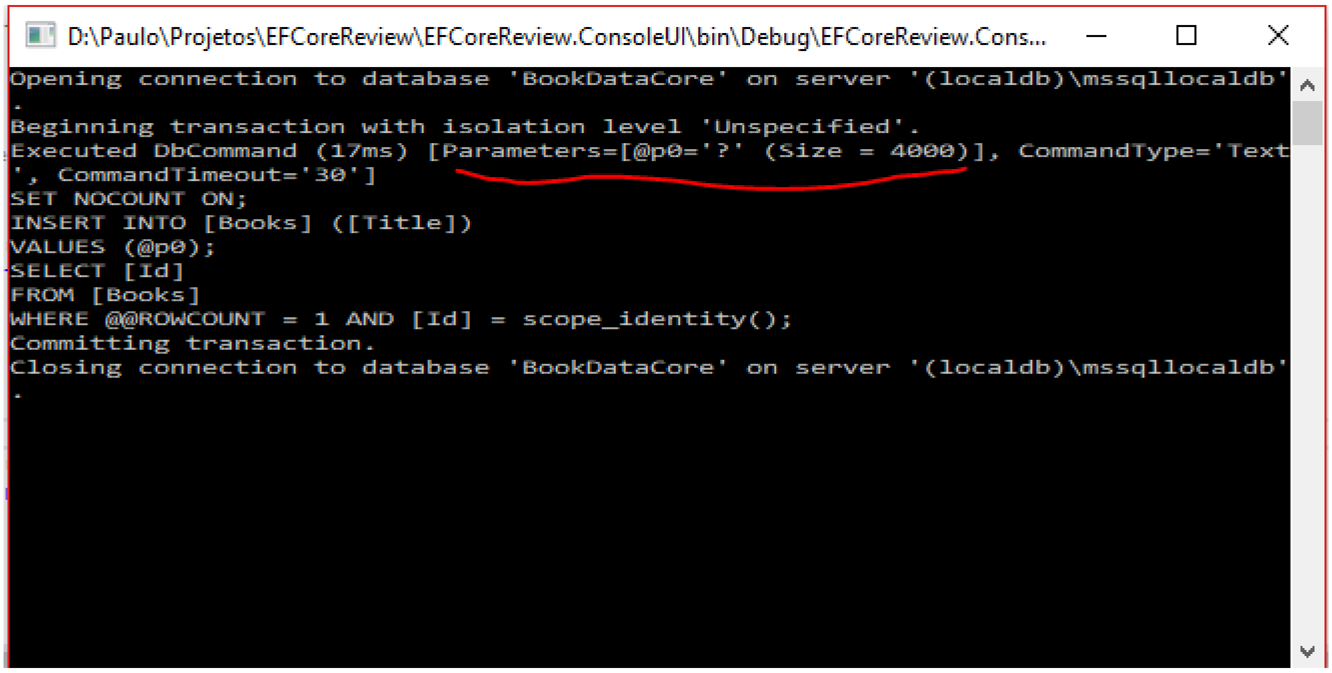
**List 7**. Logging to console

Now we have a logging implementation we must configure our context to use it. Since this is a demonstration we register the logging in the InsertBook method as well (List 8).

```
        private static void InsertBook()
        {
            var book = new Book() { Title = "C# 7 and .NET Core: Moder Cross-Platform Development"
};
            using (var context = new BookContext())
            {
                context.GetService<ILoggerFactory>().AddProvider(new MyLoggerProvider());
                context.Books.Add(book);
                context.SaveChanges();
            }
        }
```

**List 8**. Configuring the context

As we can see, we use the method GetService from DbContext to register the logging implementation. Now we can run the console application (Figure 7).



D:\Paulo\Projetos\EFCoreReview\EFCoreReview.ConsoleUI\bin\Debug\EFCoreReview.Cons...

```
Opening connection to database 'BookDataCore' on server '(localdb)\mssqllocaldb'
.
Beginning transaction with isolation level 'Unspecified'.
Executed DbCommand (17ms) [Parameters=[@p0='?' (Size = 4000)], CommandType='Text
', CommandTimeout='30']
SET NOCOUNT ON;
INSERT INTO [Books] ([Title])
VALUES (@p0);
SELECT [Id]
FROM [Books]
WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
Committing transaction.
Closing connection to database 'BookDataCore' on server '(localdb)\mssqllocaldb'
.
```

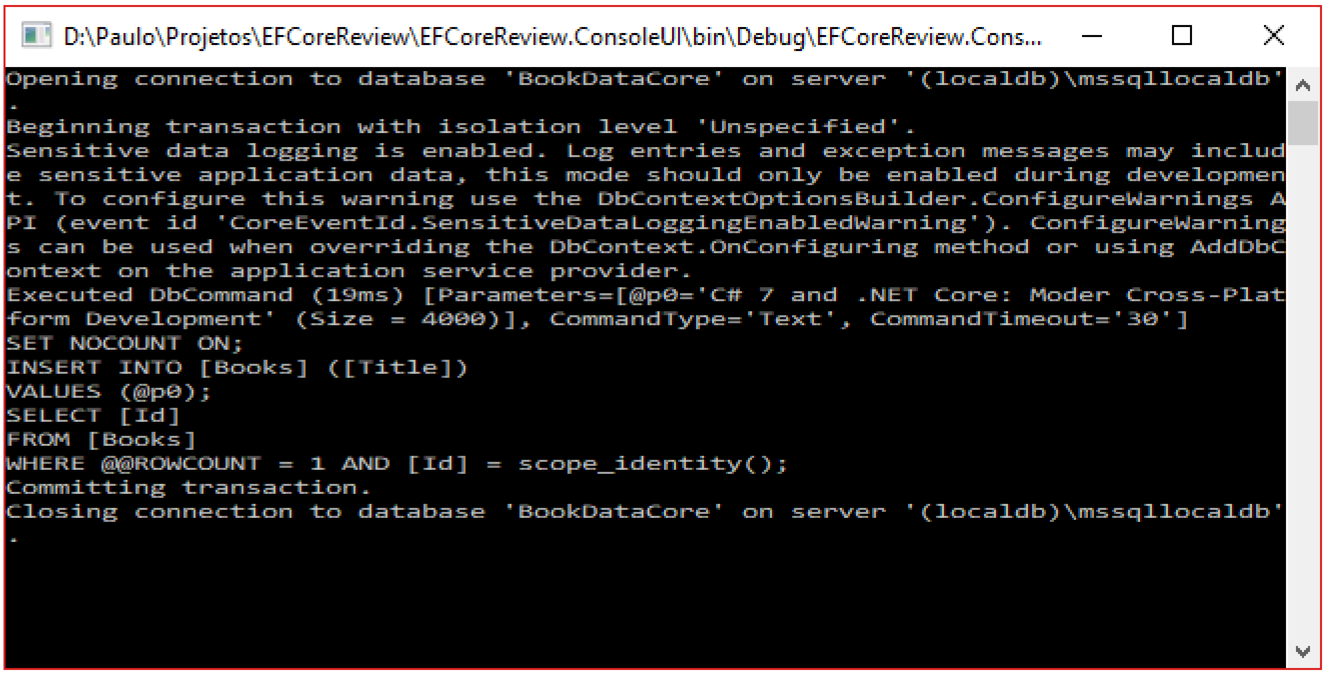
**Figure 7**. Logging working

As we can see in Figure 7 the logging hides sensitive information, for example he parameter names and values. We can change that configuring the context (List 9):

```
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer(
                "Server = (localdb)\\mssqllocaldb; Database= BookDataCore; Trusted_Connection = 
True;");
            optionsBuilder.EnableSensitiveDataLogging();
        }
```

**List 9**. Configuring logging

See the difference (Figure 8).



D:\Paulo\Projetos\EFCoreReview\EFCoreReview.ConsoleUI\bin\Debug\EFCoreReview.Cons...

```
Opening connection to database 'BookDataCore' on server '(localdb)\mssqllocaldb'
.
Beginning transaction with isolation level 'Unspecified'.
Sensitive data logging is enabled. Log entries and exception messages may includ
e sensitive application data, this mode should only be enabled during developmen
t. To configure this warning use the DbContextOptionsBuilder.ConfigureWarnings A
PI (event id 'CoreEventId.SensitiveDataLoggingEnabledWarning'). ConfigureWarning
s can be used when overriding the DbContext.OnConfiguring method or using AddDbC
ontext on the application service provider.
Executed DbCommand (19ms) [Parameters=[@p0='C# 7 and .NET Core: Moder Cross-Plat
form Development' (Size = 4000)], CommandType='Text', CommandTimeout='30']
SET NOCOUNT ON;
INSERT INTO [Books] ([Title])
VALUES (@p0);
SELECT [Id]
FROM [Books]
WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
Committing transaction.
Closing connection to database 'BookDataCore' on server '(localdb)\mssqllocaldb'
.
```

**Figure 8**. Sensitive data configured

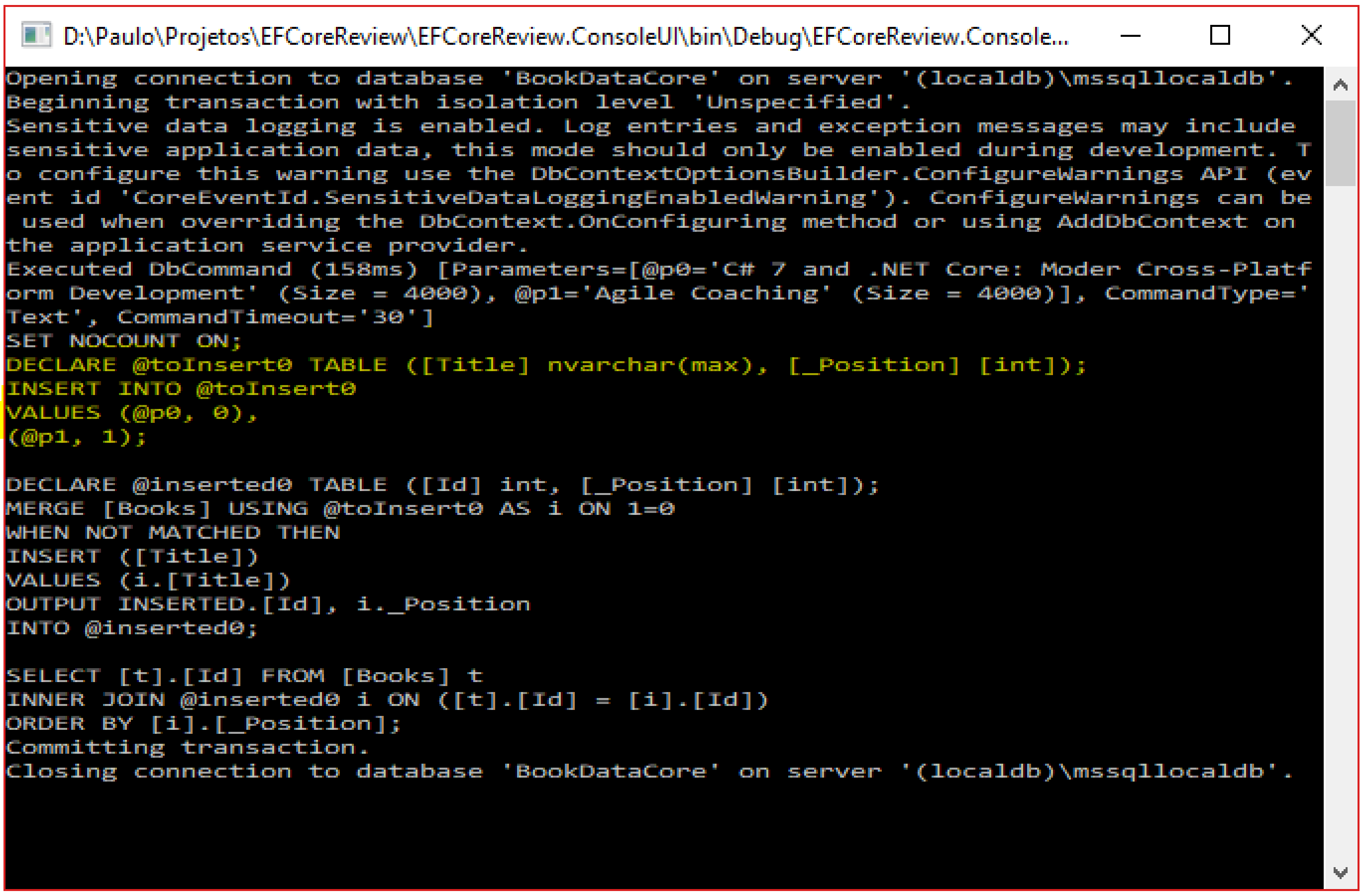
## Bulk operations

Batch operations feature was an old request from EF users, now EF Core has it (List 10).

```
        private static void InsertMultipleBooks()
        {
            var book = new Book() { Title = "C# 7 and .NET Core: Moder Cross-Platform Development"
};
            var otherbook = new Book() { Title = "Agile Coaching" };
            using (var context = new BookContext())
            {
                context.GetService<ILoggerFactory>().AddProvider(new MyLoggerProvider());
                //We can do:
                // 1. Call Add twice
                //context.Books.Add(book);
                //context.Books.Add(otherbook);
                // 2. Call AddRange and pass the objects
                //context.Books.AddRange(book, otherbook);
                // 3. Call AddRange and pass a list of Book
                context.Books.AddRange(new List<Book>() { book, otherbook });

                context.SaveChanges();
            }
        }
```

**List 10**. Adding multiple objects

Running this method produces a different response from EF 6. Check it out in Figure 9.


D:\Paulo\Projetos\EFCoreReview\EFCoreReview.ConsoleUI\bin\Debug\EFCoreReview.Console...

```
Opening connection to database 'BookDataCore' on server '(localdb)\mssqllocaldb'.
Beginning transaction with isolation level 'Unspecified'.
Sensitive data logging is enabled. Log entries and exception messages may include
sensitive application data, this mode should only be enabled during development. T
o configure this warning use the DbContextOptionsBuilder.ConfigureWarnings API (ev
ent id 'CoreEventId.SensitiveDataLoggingEnabledWarning'). ConfigureWarnings can be
 used when overriding the DbContext.OnConfiguring method or using AddDbContext on
the application service provider.
Executed DbCommand (158ms) [Parameters=[@p0='C# 7 and .NET Core: Moder Cross-Platf
orm Development' (Size = 4000), @p1='Agile Coaching' (Size = 4000)], CommandType='
Text', CommandTimeout='30']
SET NOCOUNT ON;
DECLARE @toInsert0 TABLE ([Title] nvarchar(max), [_Position] [int]);
INSERT INTO @toInsert0
VALUES (@p0, 0),
(@p1, 1);

DECLARE @inserted0 TABLE ([Id] int, [_Position] [int]);
MERGE [Books] USING @toInsert0 AS i ON 1=0
WHEN NOT MATCHED THEN
INSERT ([Title])
VALUES (i.[Title])
OUTPUT INSERTED.[Id], i._Position
INTO @inserted0;

SELECT [t].[Id] FROM [Books] t
INNER JOIN @inserted0 i ON ([t].[Id] = [i].[Id])
ORDER BY [i].[_Position];
Committing transaction.
Closing connection to database 'BookDataCore' on server '(localdb)\mssqllocaldb'.
```

**Figure 9**. Batch

With the help of SQL Server provider, some specific tables are used to manage bulk operations. The default batch size is 1,000 but we can change that too, in the BookContext (List 11):

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer(
                "Server = (localdb)\\mssqllocaldb; Database= BookDataCore; Trusted_Connection =
True;", options=> options.MaxBatchSize(30));
            optionsBuilder.EnableSensitiveDataLogging();
        }
```

**List 11**. Configuring batch size

## Querying simple objects

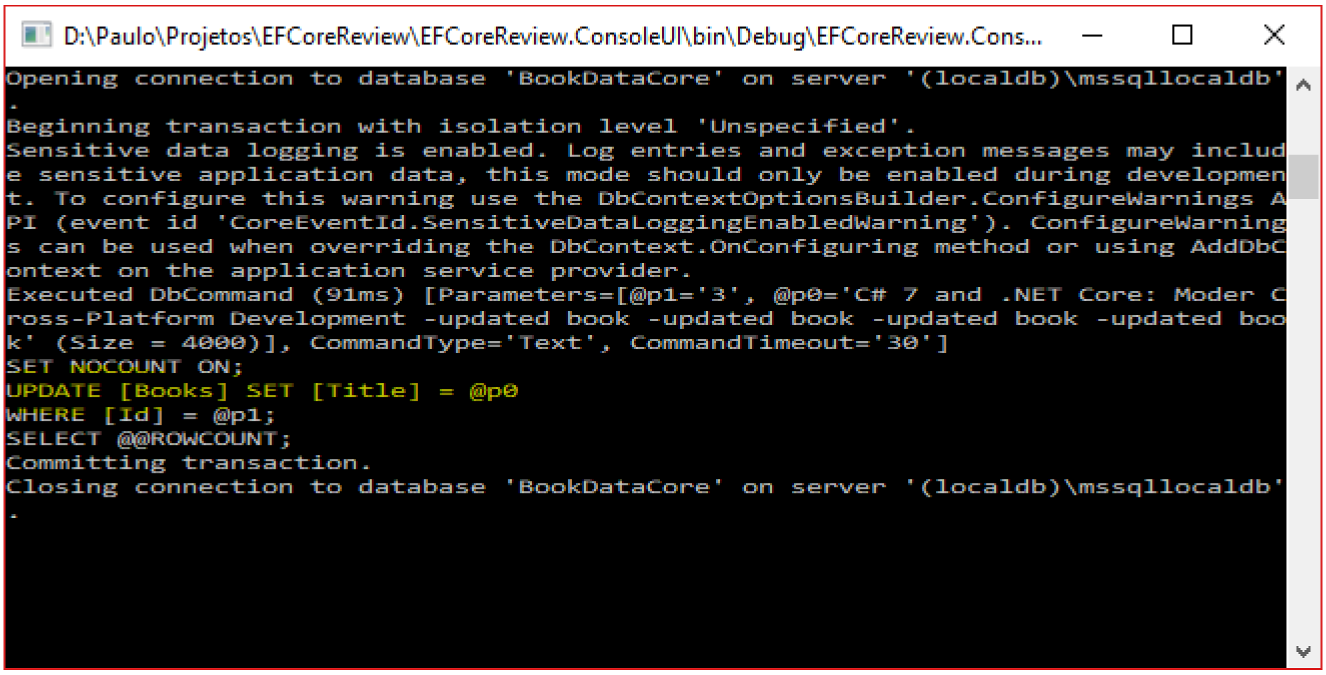
Linq to Entity execution methods: ToList(), First(), FirstOrDefault(), Single(), SingleOrDefault(),Count(), LongCount(), Min(), Max() and their Async variants. With EF Core the following methods are evaluated on client side: Last(), LastOrDefault(), Average() and their Async() variants.

## Updating objects

The method RetrieveAndUpdateBook is quite simple, and because the context is keeping track of the object retrieved, is easy for EF Core to update efficiently the object (List 12) (Figure 10).

```
        private static void RetrieveAndUpdateBook()
        {
            var context = new BookContext();
            var book = context.Books.FirstOrDefault();
            book.Title += " -updated book";
            context.SaveChanges();
        }
```

**List 12**. Update



```
D:\Paulo\Projetos\EFCoreReview\EFCoreReview.ConsoleUI\bin\Debug\EFCoreReview.Cons...
Opening connection to database 'BookDataCore' on server '(localdb)\mssqllocaldb'
.
Beginning transaction with isolation level 'Unspecified'.
Sensitive data logging is enabled. Log entries and exception messages may includ
e sensitive application data, this mode should only be enabled during developmen
t. To configure this warning use the DbContextOptionsBuilder.ConfigureWarnings A
PI (event id 'CoreEventId.SensitiveDataLoggingEnabledWarning'). ConfigureWarning
s can be used when overriding the DbContext.OnConfiguring method or using AddDbC
ontext on the application service provider.
Executed DbCommand (91ms) [Parameters=[@p1='3', @p0='C# 7 and .NET Core: Moder C
ross-Platform Development -updated book -updated book -updated book -updated boo
k' (Size = 4000)], CommandType='Text', CommandTimeout='30']
SET NOCOUNT ON;
UPDATE [Books] SET [Title] = @p0
WHERE [Id] = @p1;
SELECT @@ROWCOUNT;
Committing transaction.
Closing connection to database 'BookDataCore' on server '(localdb)\mssqllocaldb'
.
```

**Figure 10**. Updating object

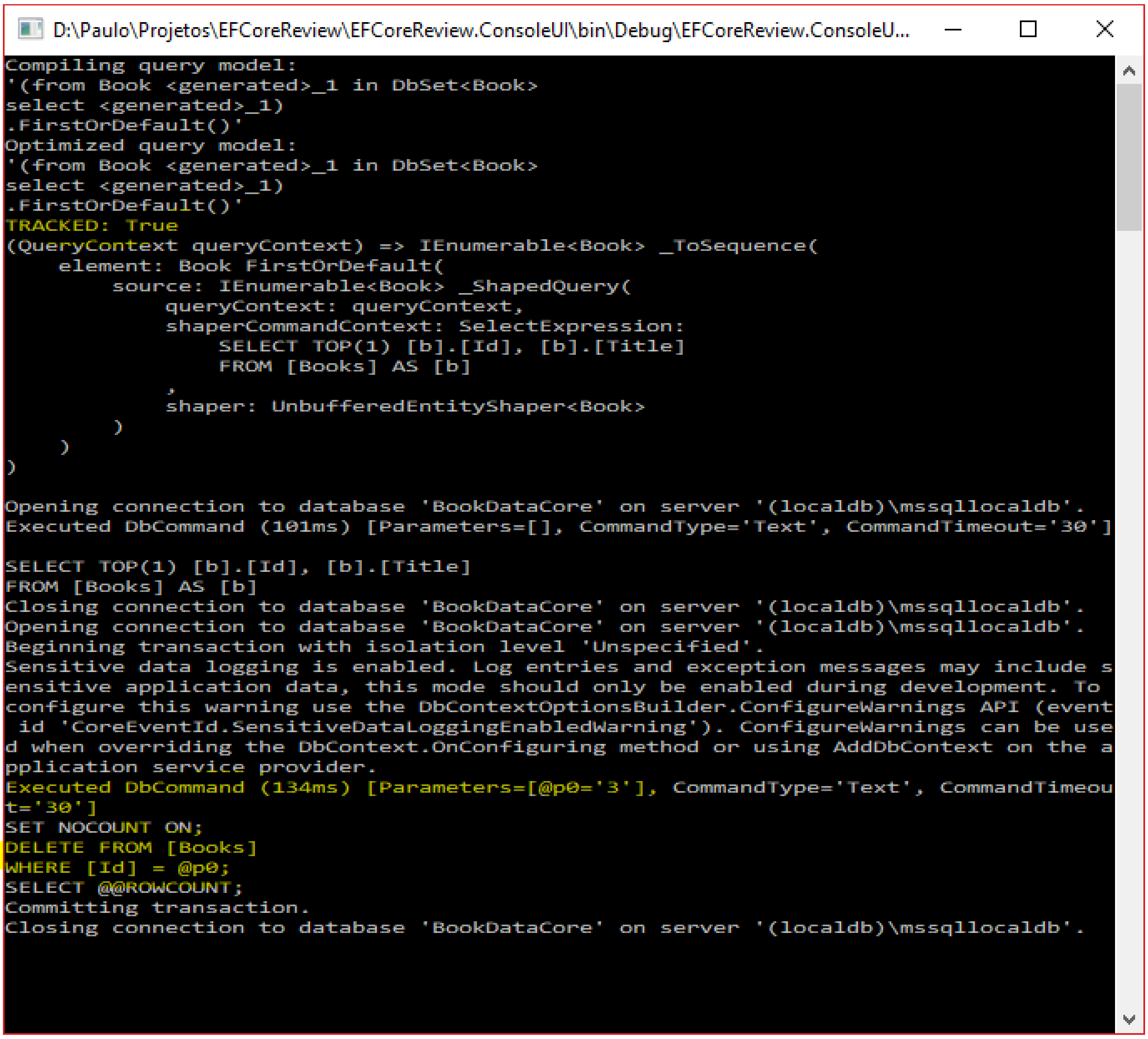
## Deleting objects

Context can **only delete tracked objects**. The method DeleteTrackedObject() show how to do it (List 13).

```
        private static void DeleteTrackedObject()
        {
            using (var context = new BookContext())
            {
                context.GetService<ILoggerFactory>().AddProvider(new MyLoggerProvider());
                var book = context.Books.FirstOrDefault();
                context.Books.Remove(book);
                // or
                //context.Remove(book);
                //context.Entry(book).State = EntityState.Deleted;
                context.SaveChanges();
            }
        }
```

**List 13.** Deleting object

As Context has information about the object, the Delete SQL command is generated properly (Figure 11).



D:\Paulo\Projetos\EFCoreReview\EFCoreReview.ConsoleUI\bin\Debug\EFCoreReview.ConsoleU...

```
Compiling query model:
'(from Book <generated>_1 in DbSet<Book>
select <generated>_1)
.FirstOrDefault()'
Optimized query model:
'(from Book <generated>_1 in DbSet<Book>
select <generated>_1)
.FirstOrDefault()'
TRACKED: True
(QueryContext queryContext) => IEnumerable<Book> _ToSequence(
    element: Book FirstOrDefault(
        source: IEnumerable<Book> _ShapedQuery(
            queryContext: queryContext,
            shaperCommandContext: SelectExpression:
                SELECT TOP(1) [b].[Id], [b].[Title]
                FROM [Books] AS [b]
            ,
            shaper: UnbufferedEntityShaper<Book>
        )
    )
)

Opening connection to database 'BookDataCore' on server '(localdb)\mssqllocaldb'.
Executed DbCommand (101ms) [Parameters=[], CommandType='Text', CommandTimeout='30']

SELECT TOP(1) [b].[Id], [b].[Title]
FROM [Books] AS [b]
Closing connection to database 'BookDataCore' on server '(localdb)\mssqllocaldb'.
Opening connection to database 'BookDataCore' on server '(localdb)\mssqllocaldb'.
Beginning transaction with isolation level 'Unspecified'.
Sensitive data logging is enabled. Log entries and exception messages may include s
ensitive application data, this mode should only be enabled during development. To
configure this warning use the DbContextOptionsBuilder.ConfigureWarnings API (event
 id 'CoreEventId.SensitiveDataLoggingEnabledWarning'). ConfigureWarnings can be use
d when overriding the DbContext.OnConfiguring method or using AddDbContext on the a
pplication service provider.
Executed DbCommand (134ms) [Parameters=[@p0='3'], CommandType='Text', CommandTimeou
t='30']
SET NOCOUNT ON;
DELETE FROM [Books]
WHERE [Id] = @p0;
SELECT @@ROWCOUNT;
Committing transaction.
Closing connection to database 'BookDataCore' on server '(localdb)\mssqllocaldb'.
```

**Figure 11**. Delete log

Delete operations made in sequence are now **batched**.

For disconnected scenarios, EF will attach the object and mark it for deletion.

## Using SQL sentences

EF Core has a new method: *DbSet.FromSql().* It allows passing SQL sentences, stored procedures, allows parameters, must return full types at this moment, and result set column names as the mapped names. Also, returns flat data only. See RunSqlSentence method (List 14).

```
private static void RunSQLSentence()
{
    using (var context = new BookContext())
    {
        var books = context.Books.FromSql("select * from Books").ToList();
        books.ForEach(x => Console.WriteLine(x.Title));
    }
}
```

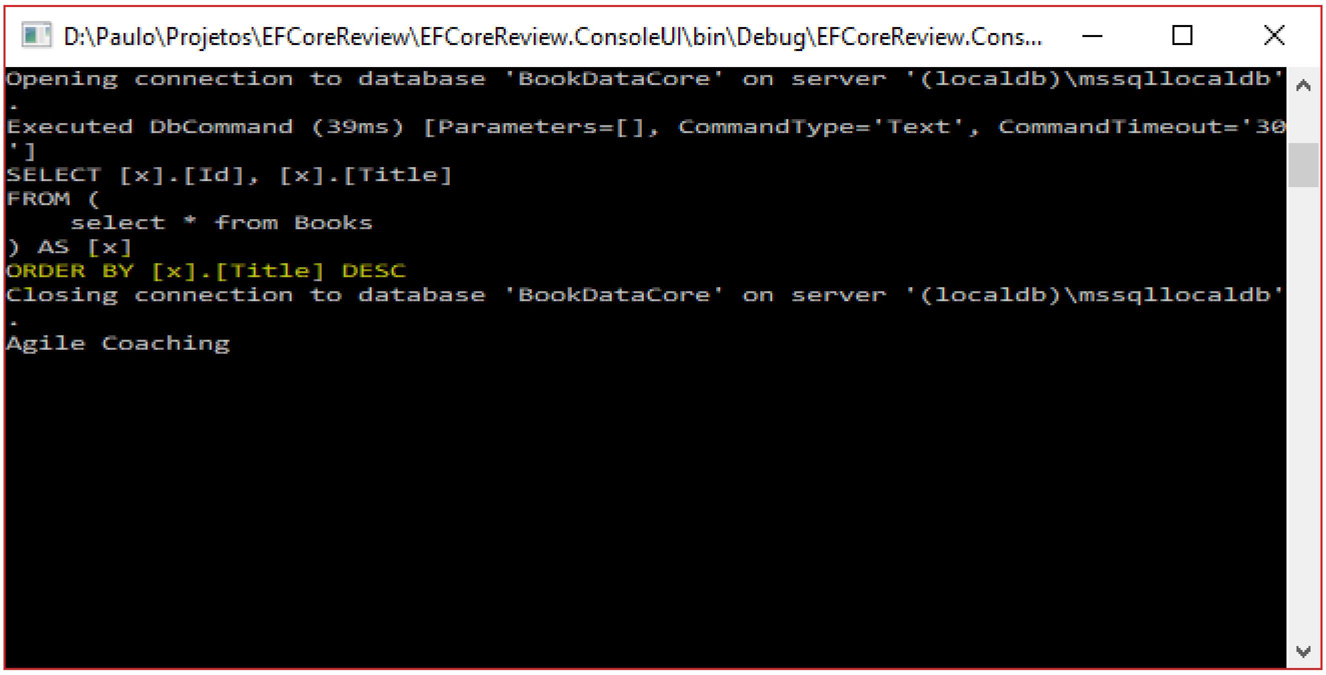
**List 14**. Running SQL

What happen if we want to sort the result set like this (List 15):

```
private static void RunSQLSentence()
{
    using (var context = new BookContext())
    {
        var books = context.Books.FromSql("select * from Books")
            .OrderByDescending(x => x.Title)
            .ToList();
        books.ForEach(x => Console.WriteLine(x.Title));
    }
}
```

**List 15**. Sorting

See the result in Figure 12.


D:\Paulo\Projetos\EFCoreReview\EFCoreReview.ConsoleUI\bin\Debug\EFCoreReview.Cons...

```
Opening connection to database 'BookDataCore' on server '(localdb)\mssqllocaldb'
.
Executed DbCommand (39ms) [Parameters=[], CommandType='Text', CommandTimeout='30
']
SELECT [x].[Id], [x].[Title]
FROM (
    select * from Books
) AS [x]
ORDER BY [x].[Title] DESC
Closing connection to database 'BookDataCore' on server '(localdb)\mssqllocaldb'
.
Agile Coaching
```

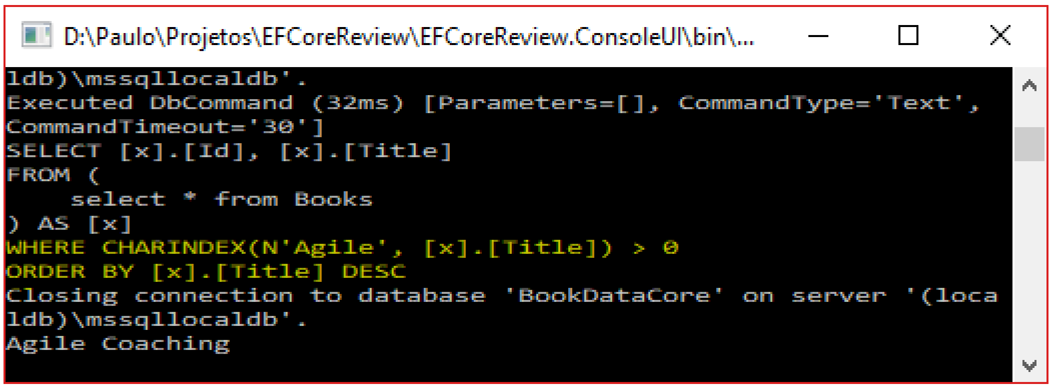
**Figure 12**. Result

EF Core figured out that order by was possible to be **translated to SQL** and did it. In EF 6 that order by would be executed on client side, in memory.

EF Core is smart enough to make this happen also when **filtering** data, List 16 and Figure 13.

```
private static void RunSQLSentence()
{
    using (var context = new BookContext())
    {
        context.GetService<ILoggerFactory>().AddProvider(new MyLoggerProvider());
        var books = context.Books.FromSql("select * from Books")
            .Where(x => x.Title.Contains("Agile"))
            .OrderByDescending(x => x.Title)
            .ToList();
        books.ForEach(x => Console.WriteLine(x.Title));
    }
}
```

**List 16**. Filtering


D:\Paulo\Projetos\EFCoreReview\EFCoreReview.ConsoleUI\bin\...

```
ldb)\mssqllocaldb'.
Executed DbCommand (32ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
SELECT [x].[Id], [x].[Title]
FROM (
    select * from Books
) AS [x]
WHERE CHARINDEX(N'Agile', [x].[Title]) > 0
ORDER BY [x].[Title] DESC
Closing connection to database 'BookDataCore' on server '(loca
ldb)\mssqllocaldb'.
Agile Coaching
```

**Figure 13**. Result

It's possible to execute non-query commands. For that we use the *DbContext.Database.ExecuteSqlCommand()* which is the same as earlier EF versions.

## Using custom functions

EF Core can understand custom functions in Linq To Entity.  Check it out in List 17.

```
private static string ReverseStr(string value)
{
    var result = value.AsEnumerable();
    return string.Concat(result.Reverse());
}

private static void QueryUsingFunctions()
{
    using (var context = new BookContext())
    {
        var books = context.Books
            .Select(x => new { newTitle = ReverseStr(x.Title) })
            .ToList();
        books.ForEach(x => Console.WriteLine(x.newTitle));
    }
}
```

**List 17**. Using custom function

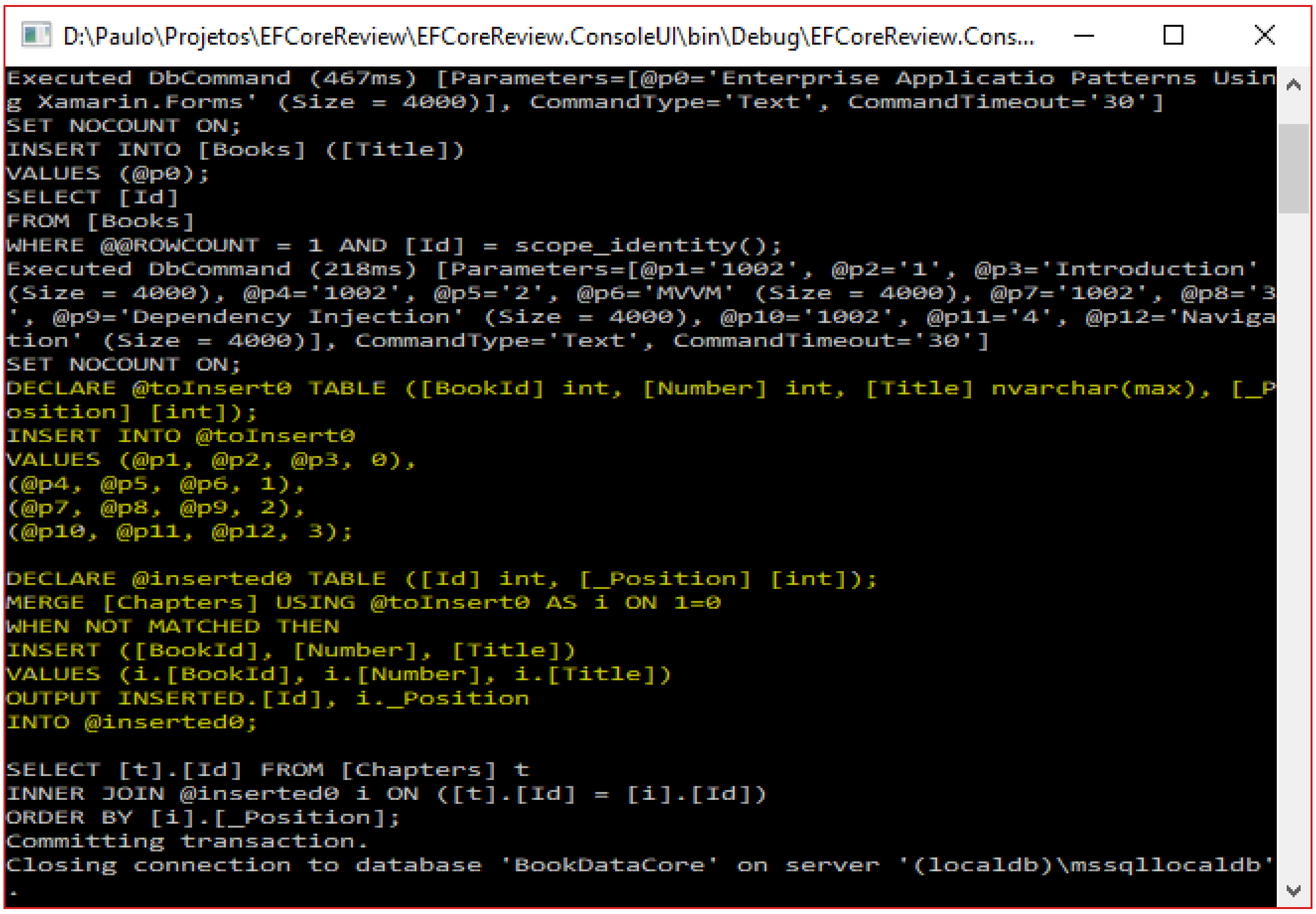
We've created a custom function ReverseStr and used it in the Select. Now EF Core get the results from the DB and iterate through them applying the custom function in memory. So, we must keep an eye on performance here.

## More complex operations

When saving a 1..N relation, EF Core **now batches the insert operations** in the dependent side. See the method AddBook (List 18) and its response (Figure 14). We create a new book with its chapters.

```
private static void AddBook()
{
    var book = new Book()
    {
        Title = "Enterprise Applicatio Patterns Using Xamarin.Forms",
        Chapters = new List<Chapter>
        {
            new Chapter() {Number = 1, Title = "Introduction"},
            new Chapter() {Number = 2, Title = "MVVM"},
            new Chapter() {Number = 3, Title = "Dependency Injection"},
            new Chapter() {Number = 4, Title = "Navigation"}
        }
    };

    using (var context = new BookContext())
    {
        context.GetService<ILoggerFactory>().AddProvider(new MyLoggerProvider());
        context.Add(book);
        context.SaveChanges();
    }
}
```

**List 18**. Adding books


D:\Paulo\Projetos\EFCoreReview\EFCoreReview.ConsoleUI\bin\Debug\EFCoreReview.Cons...

```
Executed DbCommand (467ms) [Parameters=[@p0='Enterprise Applicatio Patterns Usin
g Xamarin.Forms' (Size = 4000)], CommandType='Text', CommandTimeout='30']
SET NOCOUNT ON;
INSERT INTO [Books] ([Title])
VALUES (@p0);
SELECT [Id]
FROM [Books]
WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
Executed DbCommand (218ms) [Parameters=[@p1='1002', @p2='1', @p3='Introduction'
(Size = 4000), @p4='1002', @p5='2', @p6='MVVM' (Size = 4000), @p7='1002', @p8='3
', @p9='Dependency Injection' (Size = 4000), @p10='1002', @p11='4', @p12='Naviga
tion' (Size = 4000)], CommandType='Text', CommandTimeout='30']
SET NOCOUNT ON;
DECLARE @toInsert0 TABLE ([BookId] int, [Number] int, [Title] nvarchar(max), [_P
osition] [int]);
INSERT INTO @toInsert0
VALUES (@p1, @p2, @p3, 0),
(@p4, @p5, @p6, 1),
(@p7, @p8, @p9, 2),
(@p10, @p11, @p12, 3);

DECLARE @inserted0 TABLE ([Id] int, [_Position] [int]);
MERGE [Chapters] USING @toInsert0 AS i ON 1=0
WHEN NOT MATCHED THEN
INSERT ([BookId], [Number], [Title])
VALUES (i.[BookId], i.[Number], i.[Title])
OUTPUT INSERTED.[Id], i._Position
INTO @inserted0;

SELECT [t].[Id] FROM [Chapters] t
INNER JOIN @inserted0 i ON ([t].[Id] = [i].[Id])
ORDER BY [i].[_Position];
Committing transaction.
Closing connection to database 'BookDataCore' on server '(localdb)\mssqllocaldb'
.
```

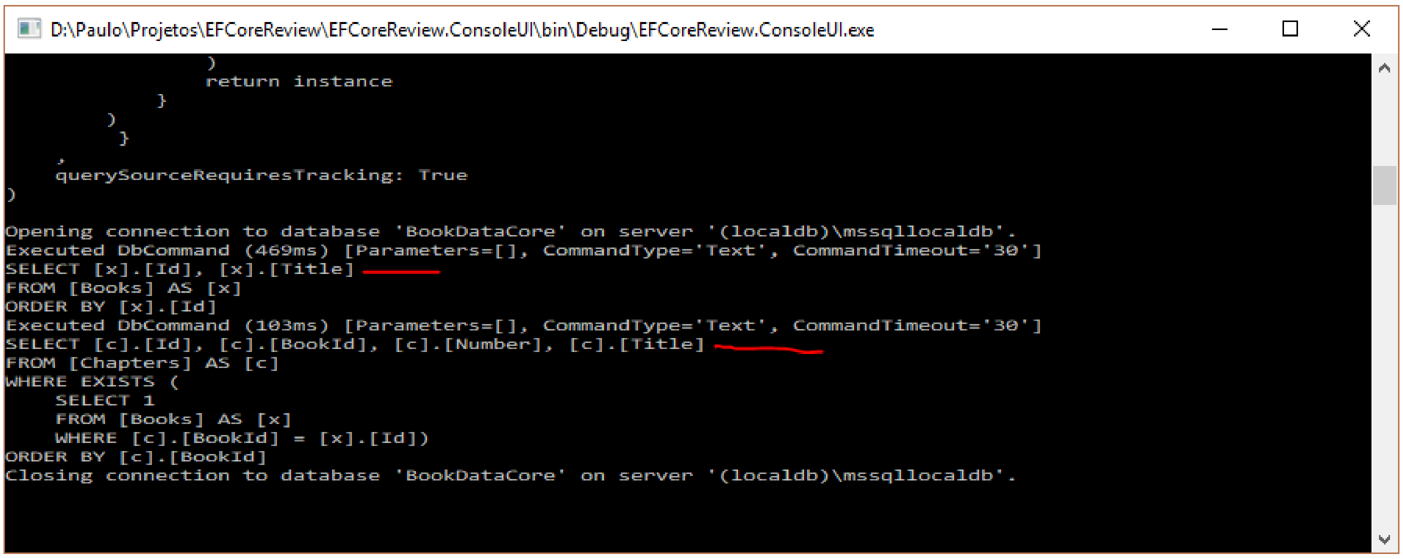
**Figure 14**. Insert batched

## Loading related classes

In EF Core 1.1  we can load related data using Include() method or explicit Load. Lazy loading is not available yet and projections are not a production ready feature. There is a change in EF Core about the generated SQL for Include(). See the method EagerUsingInclude() (List 19) and its response (Figure 15).

```
private static void EagerUsingInclude()
{

    using (var context = new BookContext())
    {
        context.GetService<ILoggerFactory>().AddProvider(new MyLoggerProvider());
        var bookList = context.Books
            .Include(x => x.Chapters)
            .ToList();
    }
}
```

**List 19**. Eager using include



```
D:\Paulo\Projetos\EFCoreReview\EFCoreReview.ConsoleUI\bin\Debug\EFCoreReview.ConsoleUI.exe
                )
                return instance
            }
        )
         }
    ,
    querySourceRequiresTracking: True
)

Opening connection to database 'BookDataCore' on server '(localdb)\mssqllocaldb'.
Executed DbCommand (469ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT [x].[Id], [x].[Title]
FROM [Books] AS [x]
ORDER BY [x].[Id]
Executed DbCommand (103ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT [c].[Id], [c].[BookId], [c].[Number], [c].[Title]
FROM [Chapters] AS [c]
WHERE EXISTS (
    SELECT 1
    FROM [Books] AS [x]
    WHERE [c].[BookId] = [x].[Id])
ORDER BY [c].[BookId]
Closing connection to database 'BookDataCore' on server '(localdb)\mssqllocaldb'.
```

**Figure 15.** Log for eager

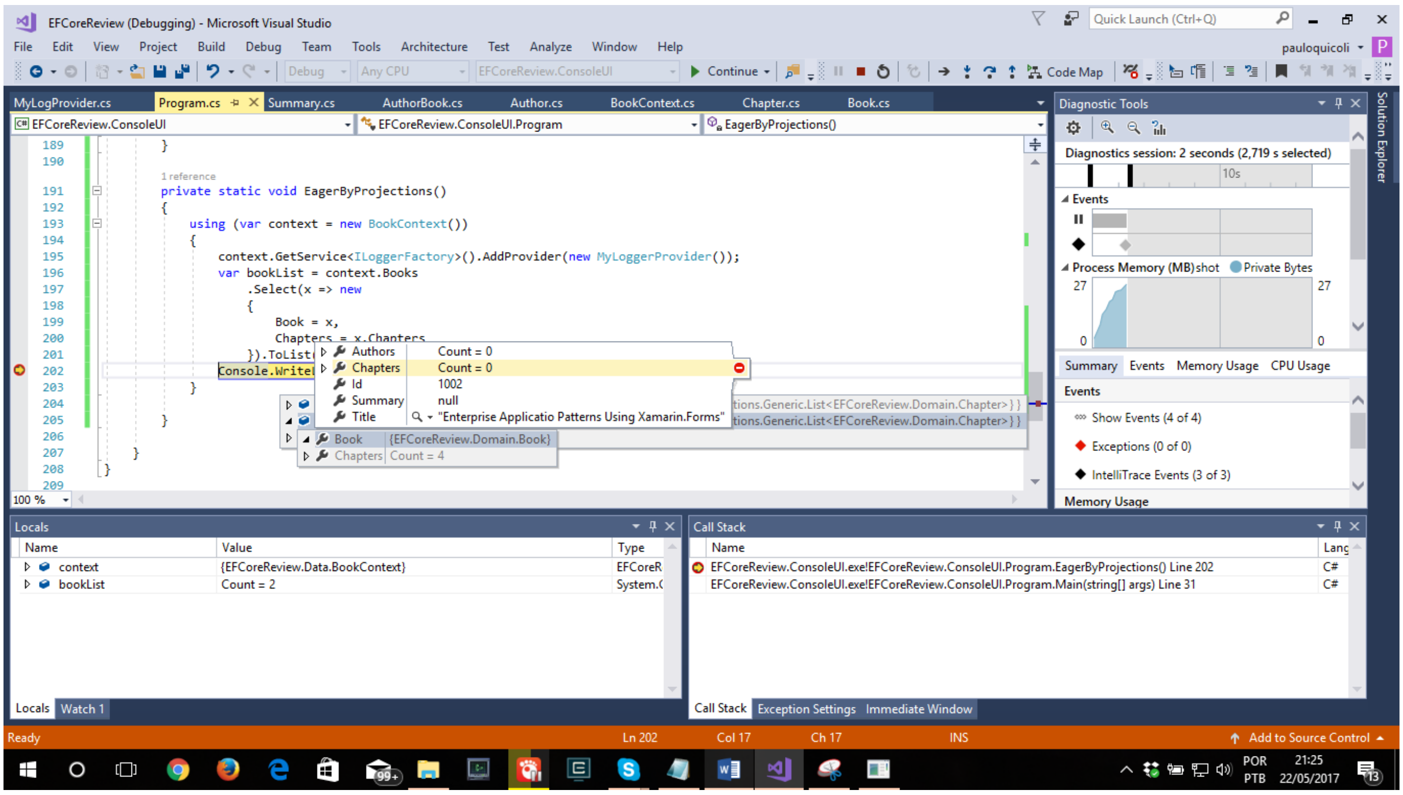
EF 6 generated a big SQL sentence returning redundant and flattered data, EF Core fix data creating multiple queries. We can see two SQL statements, one for Books and the second for Chapter collection. Once data is returned, the Context creates the appropriate graph.

Projections work as expected when they are for single types. When they involve relationships, the result is the expected but a N+1 side effect is created.  If we use projections to create an Eager Loading, the result is not the expected when comparing to EF 6. Check  EagerByProjection() method (List 20).

```
private static void EagerByProjections()
{
    using (var context = new BookContext())
    {
        context.GetService<ILoggerFactory>().AddProvider(new MyLoggerProvider());
        var bookList = context.Books
            .Select(x => new
            {
                Book = x,
                Chapters = x.Chapters
            }).ToList();
        Console.WriteLine(bookList.Count());
    }
}
```

**List 20.** Using projections to eager data

The Figure 16 shows that we got the chapters correctly. There is 4 chapters for that book, but when we look inside the returned book instance, the Chapters property is empty. It happens because EF Core wasn't tracking Chapter entity. It's a known bug (https://github.com/aspnet/entityframework/issues/7131).



```
        }

        1 reference
        private static void EagerByProjections()
        {
            using (var context = new BookContext())
            {
                context.GetService<ILoggerFactory>().AddProvider(new MyLoggerProvider());
                var bookList = context.Books
                    .Select(x => new
                    {
                        Book = x,
                        Chapters = x.Chapters
                    }).ToList(
                Console.Write
            }

        }

    }
}
```

```
Authors     Count = 0
Chapters    Count = 0
Id          1002
Summary     null
Title       "Enterprise Applicatio Patterns Using Xamarin.Forms"
Book        {EFCoreReview.Domain.Book}
Chapters    Count = 4
```

Locals

| Name | Value | Type |
|---|---|---|
| context | {EFCoreReview.Data.BookContext} | EFCoreR |
| bookList | Count = 2 | System.( |

Call Stack

| Name | Lang |
|---|---|
| EFCoreReview.ConsoleUI.exe!EFCoreReview.ConsoleUI.Program.EagerByProjections() Line 202 | C# |
| EFCoreReview.ConsoleUI.exe!EFCoreReview.ConsoleUI.Program.Main(string[] args) Line 31 | C# |

**Figure 16**. Wrong behavior

Projections that filter on the children (sub selects), will not query children unless ToList() or other Linq execute methods are used (List 21)

```
private static void ProjectionSubQueries()
{
    using (var context = new BookContext())
    {
        context.GetService<ILoggerFactory>().AddProvider(new MyLoggerProvider());
        var bookList = context.Books
            .Select(x => new
            {
                Book = x,
                Chapters = x.Chapters.Where(c => c.Number == 2).ToList()
            }).ToList();
        Console.WriteLine(bookList.Count());
    }
}
```

**List 21**. Sub select

The Fix for that right now is to use explicit loading. See ExplicitLoading() method (List 22).

```
        private static void ExplictLoading()
        {
            using (var context = new BookContext())
            {
                context.GetService<ILoggerFactory>().AddProvider(new MyLoggerProvider());
                var book = context.Books.LastOrDefault();
                context.Entry(book).Collection(x => x.Chapters)
                    .Query()
                    .Where(q => q.Number == 2)
                    .Load();
                Console.WriteLine(book.Id);
            }
        }
```

**List 22**. Fix

The syntax maybe isn't what we expected, but it works. I can filter a collection.

## Disconnected graphs

EF Core behavior **changed** from EF 6. See the following commands from DbSet/DbContext:

- Add or AddRange will mark all objects Added, regardless of key values.
- Attach or AttachRange will mark empty store-generated keys as Added, but Root (even with key values) are marked unchanged.
- Update or UpdateRange will mark empty store-generated keys as Added and Root and rest modified.
- Remove or RemoveRange will mark only Root as Deleted.

### Tracking Graphs with Entry.State

If we use Entry(objectGraph).State = EntityState **only the root object is set to the specified state**. Rest of the graph is ignored. This allows us to change the state of a root without change its graph. With EF 6 and prior it wasn't possible.

### New *TrackGraph()* method

EF Core introduces a new DbContext.ChangeTracker method:

*TrackGraph(graph, function)*

For example: *DbContext.ChangeTracker.TrackGraph(graph, x => x.State = EntityState.Attach)*

EF Core will set state for each element of the graph.

### Shadow state properties

EF Core introduces a new feature called Shadow State properties. They are properties that exist only in the Model, not in the classes. An example of use would be a request for adding properties to keep the last modified date of an object. This property can exist only in the model and maybe not in the classes. See List 23.

```
        protected override void OnModelCreating(ModelBuilder modelbuilder)
        {
            modelbuilder.Entity<AuthorBook>()
                .HasKey(s => new { s.AuthorId, s.BookId });

            modelbuilder.Entity<Book>().Property<DateTime>("LastModified");
        }
```

Here are adding an LastModified property for one entity only, but we can do it for all:

```
        protected override void OnModelCreating(ModelBuilder modelbuilder)
        {
            modelbuilder.Entity<AuthorBook>()
                .HasKey(s => new { s.AuthorId, s.BookId });

            //modelbuilder.Entity<Book>().Property<DateTime>("LastModified");

            foreach (var entType in modelbuilder.Model.GetEntityTypes())
            {
                modelbuilder.Entity(entType.Name).Property<DateTime>("LastModified");
            }
        }
```

**List 23.** Shadow state properties

As shadow state properties are not part of classes, we interact with them in the Context. In our scenery whenever an object is saved, the LastModified property should be updated. For that we override the SaveChanges method (List 24).

```
        public override int SaveChanges()
        {
            foreach (var item in ChangeTracker.Entries()
                .Where (x => x.State == EntityState.Added || x.State == EntityState.Modified))
            {
                item.Property("LastModified").CurrentValue = DateTime.Now;
            }

            return base.SaveChanges();
        }
```

**List 24**. Using the shadow state properties

Remember, to update the database schema, we need create a new migration before.

**InMemoryProvider**

InMemoryProvider is a new provider that makes easier testing EF Core because it can be used in place of a database provider, so our tests will not hit a database.

As EF Core is modular now, to use InMemoryProvider we must install the Microsoft.EntityFrameworkCore.InMemory and change our DbContext (List 25).

```
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            //optionsBuilder.UseSqlServer(
            //    "Server = (localdb)\\mssqllocaldb; Database= BookDataCore; Trusted_Connection =
True;", options=> options.MaxBatchSize(30));
            //optionsBuilder.EnableSensitiveDataLogging();
            optionsBuilder.UseInMemoryDatabase();
        }
```

**List 25.** Configuring InMemoryProvider

**Source code:** https://github.com/quicoli/EFCore1.1-SimpleReview