PROGRAMAÇÃO I

AULA 5

Prof. Alan Matheus Pinheiro Araya

CONVERSA INICIAL

Nesta aula, vamos explorar as principais formas de acesso a bancos de dados do .NET, utilizando *frameworks* de mapeamento objeto-relacional (ORM) e também bibliotecas como o Dapper, para que você possa compreender os seus principais aspectos e seja um(a) desenvolvedor(a) ainda mais completo(a).

Interações com bancos de dados no C# são muito simples e já estão estáveis há muitos anos, sendo uma linguagem muito fluente para lidar com acesso a estruturas de dados, em especial quando podemos combinar o poder da consulta integrada à linguagem (Linq) em *queries* de linguagem estruturada de consultas (SQL) ou mesmo não SQL (NoSQL), em sistemas gerenciadores de bando de dados (SGBDs). Ao longo desta aula, utilizaremos o banco **MySQL versão 8.0.25**.

TEMA 1 – INTRODUÇÃO AO ENTITY FRAMEWORK (EF)

O .NET possui alguns *frameworks* para operações com bancos de dados. Os mais conhecidos são o Entity Framework (EF), da própria Microsoft, e o NHibernate, um *fork* do Hibernate em Java. O EF é uma excelente opção para seu desenvolvimento, pois ele traz consigo a robustez de um *framework* maduro e a performance necessária para a maior parte das aplicações comerciais.

O EF é um *framework* de ORM, o que significa que ele pode traduzir a modelagem de orientação a objetos (OO) para a modelagem relacional de bancos de dados, como MySQL, PostgreSQL, SQL Server, Oracle Database, entre outros. Em suas últimas versões, também foram adicionados uma série de *adapters* para que ele possa interagir com bancos não relacionais, como: MongoDB, Cassandra, CosmosDB, entre outros. Assim como o .NET foi reescrito em uma versão OpenSource, mais leve e performática, e ganhou o nome de *.NET Core*, o EF também. Você encontrará muitos materiais se referindo a ele como: *EF Core*.

Neste tema, vamos aprender como funciona e como utilizar o EF no .NET.

1.1 O que é o EF?

O EF Core é um *miniframework* específico para acesso e manipulação de **banco de dados utilizando objetos** do C#. Na prática, ele converte seus **modelos de objetos em tabelas**, **propriedades em colunas** e assim por

diante, de forma que o desenvolvedor se preocupe apenas com a modelagem de objetos e a programação toda em C#, **sem a necessidade de escrever uma instrução SQL** e deixando que o próprio EF converta suas *queries* Linq em SQL, gerencie o contexto de transações, a abertura e o fechamento de conexão com o banco etc.

O EF Core, assim como várias bibliotecas de funções específicas do .NET, está distribuído em gerenciador de pacotes NuGet (NuGet *packages*). Até agora, não foi necessário que instalássemos pacotes adicionais em nossos códigos/projetos; mas, para o uso do EF Core, isso será necessário. Recomendamos a instalação do pacote Microsoft.EntityFrameworkCore. Para saber mais sobre como instalar um pacote NuGet no Visual Studio ou Visual Code, você pode ler materiais auxiliares como o disponível neste *link*: <https://docs.microsoft.com/pt-br/nuget/quickstart/install-and-use-a-package-in-visual-studio> (Início, 2018). Você encontrará também outros materiais sobre o que são pacotes NuGet.

O Quadro 1 mostra a equivalência de estruturas de banco (tabelas, linhas etc.) no C# quando modelamos dados (relacionais) usando objetos do C#.

Quadro 1 – Comparativo entre entidades e comandos SQL para C# no EF Core

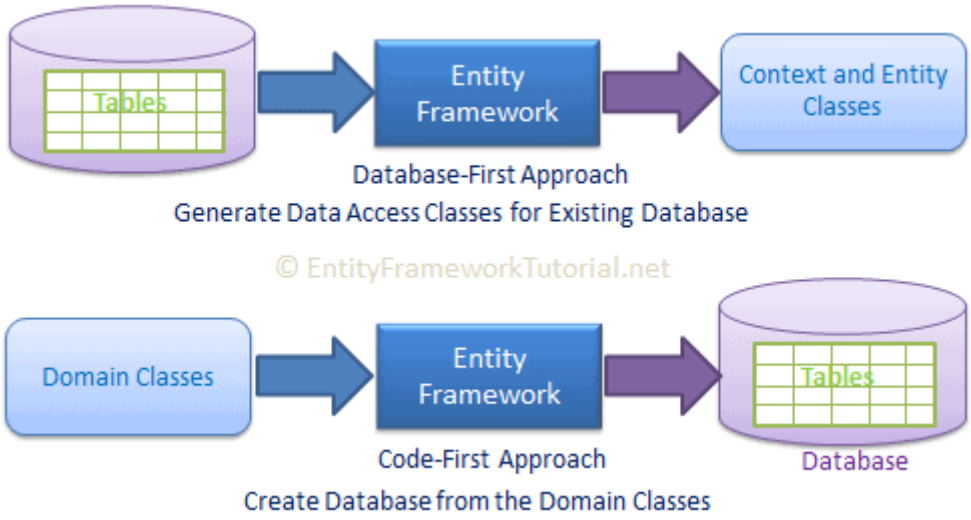
Base relacional	.NET/C#
Tabela	Classe
Colunas de tabelas	Propriedades
Linhas de tabelas	Elementos de uma coleção, p. ex.: linhas de uma List<>
Chaves primárias (*primary keys*)	Uma única instância de uma classe
Chaves estrangeiras (*foreign keys*)	Referência para outra classe
Comandos SQL, como *WHERE* (p. ex.)	Operadores Linq (Where(x=> …..))

É claro que ORMs (como o caso do EF) não são perfeitos para todos os tipos de tarefas com bancos de dados. Muitas vezes, é necessário executar um comando específico no banco, como uma *Procedure* ou uma *Function*. Para isso, o EF Core disponibiliza também métodos os quais você pode utilizar para executar uma *query* SQL e obter um retorno, podendo com isso, inclusive, mapear o retorno para um objeto ou uma lista de objetos, no C#.

Como mostra a Figura 1, existem dois modos de utilização do EF em aplicações NET Core/5/6:

1. ***Database first***: permite você criar objetos manualmente ou automaticamente, com base em tabelas já existentes no banco de dados. Essa abordagem lhe dá mais domínio sobre como as tabelas e colunas são criadas, podendo personalizar e criar atributos específicos de seu provedor de banco de dados, nessas estruturas.
2. ***Code first***: gera toda a estrutura de banco de dados: tabelas, colunas, relações, restrições (*constraints*) etc. com base em um modelo de objetos em C#.

Figura 1 – Diferença entre *code first* e *database first*



Fonte: Entity, [20--].

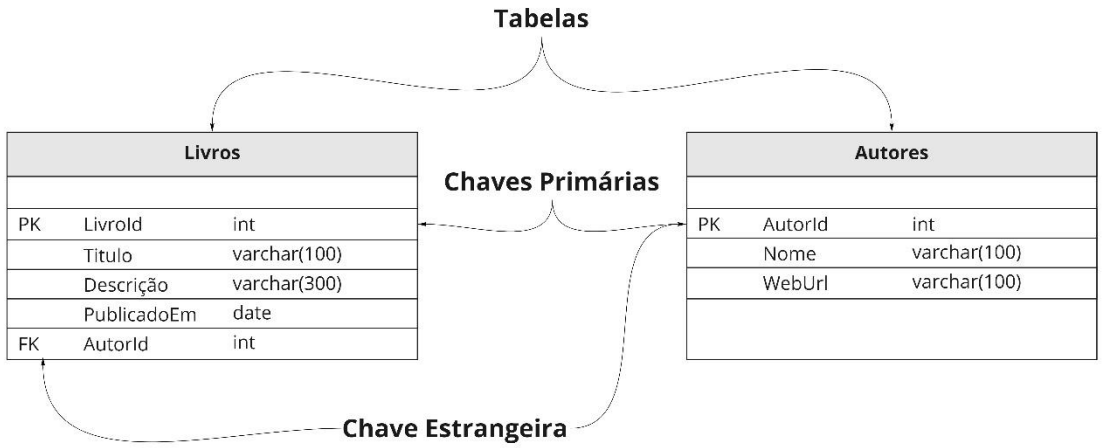
1.2 Modelagem de objetos versus modelagem relacional

É importante entendermos que não existe uma forma única de modelar todos os cenários possíveis permitidos pelas linguagens orientadas a objetos nos bancos de dados relacionais. Características elementares da OO, como herança e polimorfismo, podem ter **pouca ou nenhuma** coesão com modelos relacionais de *relational database management system* (RDBMS). **E isso não é exatamente um problema**: basta, então, entendermos o conceito de como a modelagem funciona e criar nossas **entidades** (**objetos que possuem uma tabela correspondente no banco de dados**) de forma genérica, para que se

torne possível tirar grande proveito de ferramentas como o EF, munindo de grande velocidade e robustez o desenvolvimento de nossas aplicações.

Para ilustrar o funcionamento de modelos relacionais utilizando o EF Core em um banco de dados, vamos criar um cenário de exemplo. Nesse cenário, vamos armazenar dados de livros e autores em tabelas e criar a sua respectiva equivalência em C#. A Figura 2 demonstra a modelagem de duas tabelas intituladas *Autores* e *Livros*.

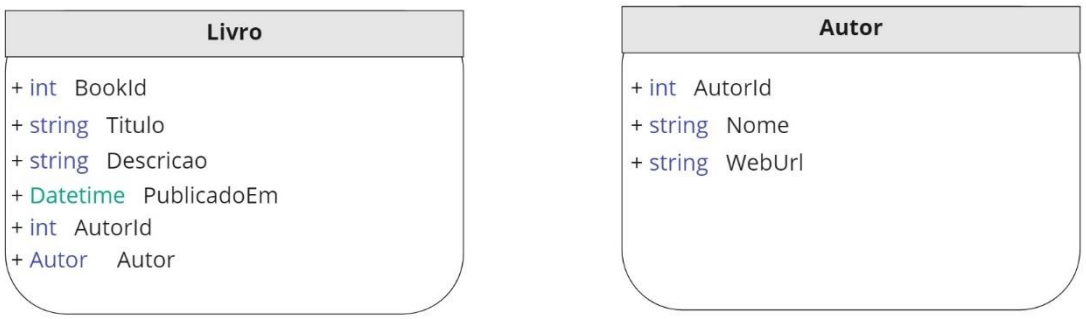
Figura 2 – Tabelas de livros e autores



Fonte: Araya, 2021.

Agora, vamos apresentar um modelo de objeto em C# que representa essa mesma estrutura e que será utilizado em nossos próximos exemplos. Observe que a classe *Livro* possuirá uma referência para a classe *Autor* e não apenas uma propriedade *AutorId*, que representa a *foreign key*.

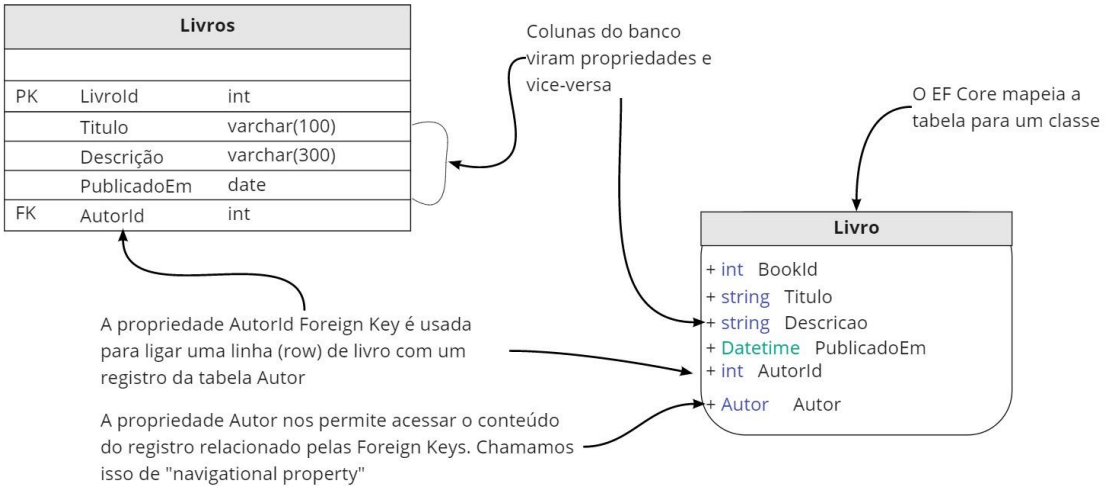
Figura 3 – Classes de livros e autores em C#



Fonte: Araya, 2021.

Observe, na Figura 4, os detalhes de como o EF mapeia as propriedades e registros da tabela para dentro de um objeto, no .NET.

Figura 4 – Mapeamento de objeto para tabela de livros, no .NET



Fonte: Araya, 2021.

A notação padrão do EF assume alguns aspectos de nomenclatura **nem sempre** muito seguidos por outros *database administrators* (DBAs) ou mesmo desenvolvedores acostumados com *database first*. Por exemplo, o nome da **tabela utilizando plural**, *Livros* e *Autores*. Essa é uma convenção **padrão de nomenclatura do EF**. Outra convenção é utilizar o nome das *primary keys* (PK) e *foreign keys* (FK) com uma combinação *<ClassName>Id*, em que *<ClassName>* é substituído pelo nome da classe em C#. Por isso, o nome da nossa propriedade de PK ou FK será: *<NomeDaClasse>Id* (Smith, 2018, p. 49).

1.3 *DbContext*

A classe *DbContext* é uma forma de o EF expor os principais métodos e propriedades para acessar e manipular o banco de dados. Essa classe atua como uma camada entre código e banco de dados. É por meio dela que o EF recebe um objeto e o transforma em um comando SQL de *INSERT INTO TABLE...*, por exemplo. É comum utilizarmos uma única *DbContext* por projeto ou por banco de dados. Mas nada nos impede de ter mais de uma *DbContext* para cada módulo ou *schema*, se assim o desejarmos. Não iremos explorar os prós e contras desse tipo de abordagem em nossas aulas, mas nas referências citadas você poderá explorar com mais detalhes esse assunto.

O *DbContext* é uma classe abstrata. Logo, precisamos herdá-la implementando alguns métodos, para usá-la em nossa aplicação. A Figura 5 fornece um exemplo de implementação de *DbContext* em uma aplicação.

Figura 5 – Exemplo de implementação da classe abstrata *DbContext*

```
public class ApplicationContext : DbContext
{
    private const string stringDeConexao = @"Server=database-
aula.cg8lcumypg6o.us-east-
2.rds.amazonaws.com;Port=3306;Database=dbaula;User Id=admin;Password=***";

    protected override void OnConfiguring(DbContextOptionsBuilder optionsB
uilder)
    {
        var serverVersion = new MySqlServerVersion(new Version(8, 0, 25));

        optionsBuilder.UseMySql(stringDeConexao, serverVersion)
                      .EnableDetailedErrors();
    }
}
```

Na Figura 5, podemos ver nossa classe *ApplicationContext* herdando de *DbContext* e configurando os dados de conexão de acordo com nosso banco de dados. Nesse exemplo, os dados de conexão estão apontando para um banco de dados MySQL rodando como *relational database service* (RDS) na Amazon Web Services (AWS). Para configurar seu próprio banco local, basta substituir o conteúdo da *tag Server*, na *string* de conexão, por *localhost*; e a *database* pelo nome de sua *database* local, assim como *user ID* pelo usuário e *password* pela senha do nosso banco de dados local.

Vale lembrar que o EF Core trabalha com vários bancos de dados. Para configurar sua conexão com MySQL, é necessário instalar o NuGet: *Pomelo.EntityFramework.MySql*. Esse NuGet provê os *providers* necessários para o EF traduzir os modelos de dados dos objetos em modelos de banco de dados e vice-versa. Para saber mais sobre os NuGets de *adapters*, leia o conteúdo disponível em: <https://docs.microsoft.com/pt-br/ef/core/dbcontext-configuration/#dbcontextoptions> (Vida, 2020).

1.4 Utilizando a *DbContext*

Para utilizar a *DbContext*, precisamos, primeiro, garantir que ela tenha um modelo criado no banco de dados. Para isso, instruímos o EF de quais classes de nosso projeto representam entidades, lembrando que toda classe que mapear

os campos de uma tabela será considerada uma entidade (*entity*, em inglês). Criamos, então, nossas entidades. Vamos seguir, inicialmente, o modelo de classes da Figura 6.

Figura 6 – Modelo de entidades para as classes *Livro* e *Autor*

```
public class Livro
{
    public int LivroId { get; set; }
    public string Titulo { get; set; }
    public string Descricao { get; set; }
    public DateTime PublicadoEm { get; set; }
    public int AutorId { get; set; }
    public Autor Autor { get; set; }
}
```

```
public class Autor
{
    public int AutorId { get; set; }
    public string Nome { get; set; }
    public string WebUrl { get; set; }
}
```

Observe que apenas definimos, no modelo da Figura 6, as propriedades conforme nosso modelo visual. Agora, vamos adicionar nossas classes dentro do contexto, para que nossa *DbContext* as entenda como entidades. Nossa classe ficará como na Figura 7.

Figura 7 – *ApplicationContext* com os *DbSet<>* das classes *Livro* e *Autor*

```
public class ApplicationContext : DbContext
{
    private const string stringDeConexao = @"Server=database-aula.cg8lcumypg6o.us-east-
2.rds.amazonaws.com;Port=3306;Database=dbaula;User Id=admin;Password=***";

    public DbSet<Livro> Livro { get; set; }
    public DbSet<Autor> Autor { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        var serverVersion = new MySqlServerVersion(new Version(8, 0, 25));

        optionsBuilder.UseMySql(stringDeConexao, serverVersion)
                      .EnableDetailedErrors();
    }
}
```

Assim, as propriedades *DbSet<T>* indicaram para a *DbContext* que desejamos que ela entenda as classes *Livro* e *Autor* como entidades.

1.5 Mapeamento do modelo

Como falamos anteriormente, a *DbContext* irá representar nosso banco de dados. E, como também vimos, cada propriedade de DbSet dentro dela irá representar uma tabela diferente do banco de dados. Você deve estar notando que estamos **construindo um modelo do banco dentro do C#, utilizando objetos**. E é exatamente isso que deve ser feito no EF. Ele irá representar uma série de operações com o banco de dados, mas sem explicitamente referenciar termos, nomenclatura ou comportamentos do banco de dados. Ele irá "falar" seu **próprio "dialeto"** de como representar seus dados, objetos e consultas. E isso é muito importante, pois a *DbContext* será **uma representação em memória, de seu banco**. Ela irá até mesmo definir regras de suas entidades, como não deixar salvar um objeto com uma propriedade, **que no banco é uma coluna que não permite *null***, sem atribuir valor a ela, por exemplo. Por isso, dizemos que a *DbContext* é uma representação em memória do nosso banco.

Para mapear nossas classes, vamos usar um modelo chamado *EntityTypeConfiguration*. Esse modelo não é o mais rápido ou o menor em termos de configuração, mas é o ideal em nível de funcionalidades oferecidas e, ao mesmo tempo, não polui nossas entidades com metadados (atributos) em cima das propriedades. Observe, na Figura 8, como é simples mapear as entidades. Basta criarmos **outra classe**, que implemente a interface *IEntityTypeConfiguration<T>*, e, dentro dela, usar a sintaxe da *builder* para definir quais **propriedades suas virarão colunas (com suas características como tamanho, nome da coluna no banco de dados, se aceitam *null*, se são PK etc.)**.

Figura 8 – Exemplo de mapeamento classe-tabela no EF

```
public class AutorMapp : IEntityTypeConfiguration<Autor>
{
    public void Configure(EntityTypeBuilder<Autor> builder)
    {
        builder.HasKey(x => x.AutorId);

        builder.Property(x => x.Nome)
               .HasMaxLength(100);

        builder.Property(x => x.WebUrl)
               .HasMaxLength(500);
    }
}
```

Então, basta adicionar o método da Figura 9 ao nosso *ApplicationContext*.

Figura 9 – Sobrescrita do método *OnModelCreating* para configurar as classes de mapeamento do EF

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.ApplyConfigurationsFromAssembly(Assembly.GetExecutingAsse
mbly());
}
```

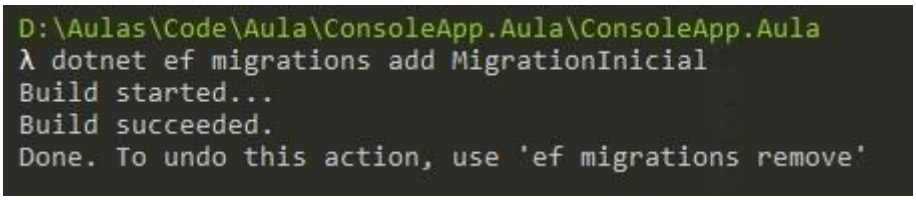
O método da Figura 9 dirá para nossa *DbContext* que ela deve procurar por classes que implementem a interface *IEntityTypeConfiguration* dentro de todas as classes de nosso projeto. Isso facilita muito o nosso dia a dia, pois organiza o código dos mapeamentos em classes separadas e evita termos que adicionar uma classe nova ao nosso *ApplicationContext* sempre que for criado um seu mapeamento.

1.6 *Migrations*

Antes de iniciarmos exemplos com as operações no banco propriamente ditas, precisamos entender, de forma macro, como utilizar o conceito de *migrations* no modelo *code first* que mencionamos anteriormente. As *migrations* são um tipo de versionamento do banco de dados. Você deve usá-las quando sua **abordagem de utilização for *code first***. Ou seja, você irá criar o banco de dados com base no seu modelo de objetos, deixando o EF criar para você as tabelas e colunas seguindo o modelo e o mapeamento que você fez. Isso agiliza muito o processo de desenvolvimento, **mas requer um pouco de cuidado para que o EF Core gere a estrutura que você efetivamente deseja**.

Para executar uma *migration*, você deve primeiro instalar o Command Line Interface (CLI) do EF Core. Basta, para tanto, abrir o seu terminal de linha de comando (no Windows, executando o comando *cmd*) e inserir o seguinte comando: *dotnet tool install --global dotnet-ef*. Caso tenha dúvidas a respeito, siga o passo a passo disponível em: <https://docs.microsoft.com/pt-br/ef/core/cli/dotnet> (Referência, 2020). Agora, navegue pelo terminal até a pasta de seu projeto (onde está o arquivo *.csproj*, usando o comando *cd <pasta>*) e digite o comando: *dotnet ef migrations add MigrationInicial* (Figura 10).

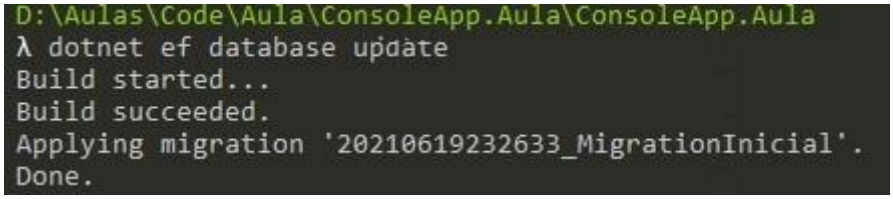
Figura 10 – Exemplo de uso do comando *add* em *Migrations* do EF



Pronto! Agora, o EF gerou uma pasta chamada *Migrations*, em seu projeto. Não a remova, pois ela será utilizada para "versionar" as atualizações do banco de dados. Recomendamos a leitura da documentação oficial sobre isso, em caso de dúvidas, disponível em: <https://docs.microsoft.com/pt-br/ef/core/managing-schemas/migrations> (Visão, 2020).

Prosseguindo, basta executar a *migration* que o EF irá ler todos os dados daquela pasta e criar as tabelas, no banco de dados, que representem seu modelo de objetos. Para isso, execute o comando: *dotnet ef database update*. Ele deve gerar a *output* que aparece na Figura 11.

Figura 11 – Exemplo de uso do comando *Update* em *Migrations* do EF

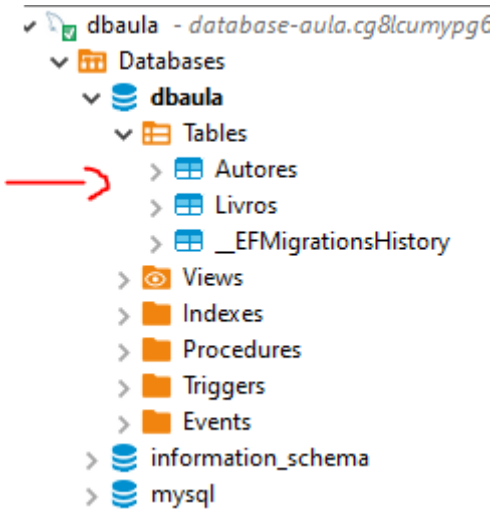


O comando *database update* sempre irá atualizar seu banco baseado na última *migration*. Assim, sempre que você modificar alguma propriedade de suas entidades ou algum detalhe do seu mapeamento, por exemplo, ao adicionar uma chave nova ou deixar um campo nulo, você deverá:

a. adicionar uma nova *migration*, por meio do comando: *dotnet ef migrations add <NomeDaMinhaMigration>*;
b. atualizar o banco de dados, por intermédio do comando: *dotnet ef database update*.

Ao final do processo, nosso banco de dados deve conter as duas tabelas representando nosso modelo e uma tabela, criada pelo EF, onde ele armazena o histórico das *migrations* aplicadas, vide Figura 12.

Figura 12 – Tabelas do banco gerado pelo EF



TEMA 2 – OPERAÇÕES DE CRUD NO EF

Vamos, então, entender como funciona a manipulação de objetos dentro do EF utilizando operações simples de *create*, *read*, *update* e *delete* (Crud). Para isso, precisamos retomar nosso modelo do Tema 1 e manipulá-lo a fim de que possamos inserir nele nossos primeiros dados. **É importante que você tenha criado a estrutura de banco de dados primeiro**, utilizando *migrations*, como explicado anteriormente; ou mesmo manualmente, desde que no modelo *database first*, dispondo as tabelas e campos diretamente na interface do seu gerenciador de banco de dados.

O EF disponibiliza os comandos de *Insert*/*Update*/*Delete* diretamente na *DbContext*. Agora, vamos estudar como manipular esses comandos. O trecho de código da Figura 13 mostra como faremos para usar a *DbContext* que criamos.

Figura 13 – Uso da *DbContext* por meio de *ApplicationContext* customizado

```
static async Task Main(string[] args)
{
    //Inicializa nosso ApplicationContext (herda de DbContext)
    ApplicationContext applicationContext = new ApplicationContext();

    //testa conexão com o banco
    var canConnect = await applicationContext.Database.CanConnectAsync();
    if (!canConnect)
    {
        //não foi possível conectar no BD, revise sua string de conexão
        return;
    }
    //Verifica se o database existe, se não existir, cria
    await applicationContext.Database.EnsureCreatedAsync();
}
```

2.1 *Create/insert*

Para realizar um *insert* no bando de dados, precisaremos utilizar o método *Add()* da *DbContext*. Você irá notar que manipular a *DbContext* é muito mais parecido com manipular uma lista de objetos em memória do C# do que com manipular, efetivamente, comandos de bancos de dados, em que você precisa passar detalhes das colunas etc. No trecho da Figura 14, o método *CriaAutores* inicializa uma lista com duas instâncias dos objetos de *Autor* e adiciona uma lista. Os nomes dos autores são homenagens aos autores dos livros por nós utilizados.

Figura 14 – Exemplo de como instanciar nossas entidades autores

```
private static List<Autor> CriaAutores()
{
    Autor autor1 = new Autor
    {
        Nome = "JON SMITH"
    };
    Autor autor2 = new Autor
    {
        Nome = "JOSEPH ALBAHARI"
    };

    var autores = new List<Autor>(2) { autor1, autor2 };
    return autores;
}
```

Observe que, no trecho da Figura 14, **não preenchemos** a propriedade *AutorId*, **mesmo ela sendo PK**. Isso porque, baseado no mapeamento que fizemos, o EF gerou uma coluna do tipo *identity column*, que incrementa automaticamente. Agora, note como faremos para inserir os nomes dos autores

no banco. Bastam, para isso, que executemos dois comandos simples: *Add()* e *SaveChangesAsync()*.

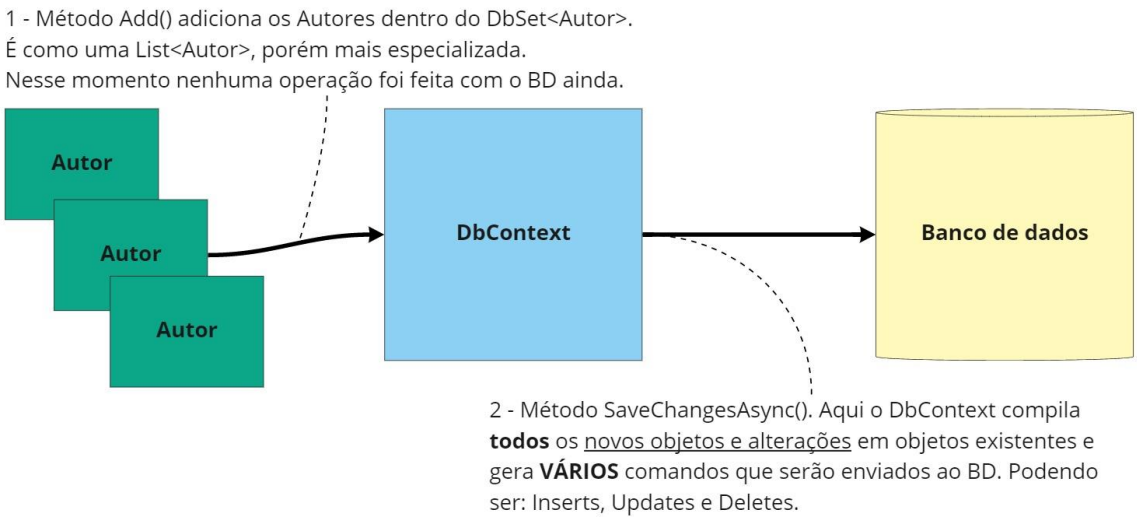
Figura 15 – Exemplo do uso do *Add* dos autores dentro de *DbSet* do *ApplicationContext*

```
//O método "CriaAutores" retorna uma lista com 2 autores
foreach (var autor in CriaAutores())
{
    applicationContext.Autores.Add(autor);
}

await applicationContext.SaveChangesAsync();
```

Os métodos da Figura 15 obedecem à lógica discriminada na Figura 16.

Figura 16 – Contexto de funcionamento do método *Add* de *DbContext*



Sempre que precisarmos adicionar um objeto novo ao banco de dados, vamos usar o método *Add()*.

2.2 *Update*

Para atualizar um dado no banco, utilizando EF, é muito simples. Primeiramente, precisamos ter o objeto mapeado pelo EF dentro do contexto, isto é, precisamos garantir que o objeto em atualização esteja sendo **rastreado** pela *DbContext*. Para fazer isso, uma das formas mais simples é recuperarmos o objeto do banco usando a instância ativa da *DbContext* e, depois disso, mudarmos os valores para os que desejamos para, então, salvar essas

alterações no banco de dados. Veja o exemplo de código da Figura 17 e vamos explicar melhor esse mencionado **rastreamento** na sequência.

Figura 17 – Exemplo de uso de *Find (async)* para atualizar uma entidade existente

```
//busca o objeto pela sua PK
//como nossa PK é uma int, estamos passando o valor da PK do autor 2 (JOSE
PH ALBAHARI)
//observe como estamos acessando o DbSet Autores (DbSet<Autor>)
var autorAlbahari = await dbContext.Autores.FindAsync(2);

//atualiza um valor de um propriedade
autorAlbahari.WebUrl = "https://www.oreilly.com/learning-paths/learning-
path-clean/8204091500000000001/";

//salva no banco de dados as mudanças
await dbContext.SaveChangesAsync();
```

O que estamos fazendo é um procedimento de recuperar do banco novamente o objeto, baseado em sua PK, e, depois, atualizando o valor de uma propriedade e novamente salvando o objeto no banco. Observe como não foi necessário executar **nenhum comando de *update*** ou outra lógica **mais complexa de SQL** para que pudéssemos atualizar valores dos objetos no banco de dados. Isso graças ao EF, que **traduz nossos objetos em comandos SQL** e os executa **toda vez que dispomos do método *SaveChangesAsync***.

Agora, vamos entender como funciona o rastreamento que comentamos. O EF mantém em memória todas as interações feitas com ele antes de enviá-las ao banco de dados. Isso quer dizer que, sempre que o EF recebe um objeto via comando *Add* ou busca um objeto via *query*, no DbSet (vamos ver outros métodos de *query*, fora o FindAsync), esse objeto fica *tracked* (rastreado), por padrão. O EF, basicamente, armazena um **clone** do seu objeto em memória e também armazena uma referência ao seu objeto real. Sempre que você usar o *SaveChnagesAsync*, o EF irá comparar as mudanças do seu objeto real com o objeto que ele clonou assim que buscou os dados do banco de dados ou o objeto original que foi adicionado à lista do DbSet. Então, se houver diferenças entre eles, o EF irá gerar uma *query* para atualizar o banco, baseado nos valores atuais do seu objeto. Esse processo chamamos de *change tracking*.

Você pode encontrar mais informações a esse respeito em documento oficial do EF Core disponível em: <https://docs.microsoft.com/pt-

br/ef/core/change-tracking/> (Controle, 2020), segundo o qual podemos ter os seguintes estados de entidades controladas pelo ChangeTracker da *DbContext*:

- ***Detached*** – as entidades não estão sendo acompanhadas pela *DbContext*.
- ***Added*** – as entidades são novas e ainda não foram inseridas no banco de dados. Isso significa que elas serão inseridas quando ocorrer comando *SaveChanges*.
- ***Unchanged*** – as entidades não foram alteradas desde que foram consultadas no banco de dados. Todas as entidades retornadas de consultas figuram inicialmente nesse estado.
- ***Modified*** – as entidades foram alteradas depois que foram consultadas no banco de dados. Isso significa que elas serão atualizadas quando se executar *SaveChanges*.
- ***Deleted*** – as entidades existem no banco de dados, mas já estão marcadas para serem excluídas quando houver *SaveChanges*.

2.3 *Remove/delete*

Para deletar um objeto do banco de dados, o processo é muito similar com realizar ações como de *update* ou *create*. Basicamente, precisamos encontrar o objeto que desejamos deletar, utilizando alguma forma de obtê-lo no banco via EF. Dessa maneira, o EF irá rastrear o objeto por meio do *change tracking*. Então, basta informamos ao DbSet que desejamos **remover** o objeto. Isso fará com que o estado desse objeto fique **marcado como *deleted*, no EF**. Mas ele **somente será deletado quando** o EF executar o comando *Delete*, no banco de dados, por meio do método *SaveChangesAsync*.

Na Figura 18, podemos ver um exemplo de chamada para o método ***remove***. Observe que criamos um autor de exemplo e, depois, escrevemos no *console* sua identidade (ID). Isso porque, se a aplicação for executada várias vezes, ela irá gerar sempre **um autor novo** e, **logo, uma nova ID**, pois a coluna *AutorId* é uma coluna do tipo *identity column* no banco de dados, o que significa que **ele irá incrementá-la sempre que um registro novo for inserido**. Note também a forma como recuperamos o novo autor, utilizando uma *query* **Linq e filtrando a busca pelo seu nome**. Vamos explorar isso melhor no próximo tópico deste tema.

Figura 18 – Exemplo de uso de *Remove* de *DbContext*

```
private static async Task DeletaAutorExemplo(ApplicationContext dbContext)
{
    //Cria um autor de exemplo para deletar
    dbContext.Autores.Add(new Autor() { Nome = "autorParaDeletar", WebUrl = "googl
e.com.br" });

    //salva no banco de dados o novo autor
    await dbContext.SaveChangesAsync();

    //busca o novo autor pelo seu nome
    var autorParaDeletar = await dbContext
                                .Autores
                                .Where(a => a.Nome == "autorParaDeletar")
                                .FirstOrDefaultAsync();
    if (autorParaDeletar != null)
    {
        Console.WriteLine($"Autor para deletar encontrado com o id: {autorParaDele
tar.AutorId}");
    }

    //marca o objeto para deleção pelo EF
    dbContext.Autores.Remove(autorParaDeletar);

    //salva no banco de dados as alteraões, neste caso,
    //o novo autor com o nome autorParaDeletar será deletado
    await dbContext.SaveChangesAsync();
 }
```

2.4 *Read/select*

Existem algumas maneiras de se realizar uma *select*, no banco de dados, utilizando o EF Core. A mais simples delas é empregar *queries* Linq para fazer uma consulta no banco, dispondo-se dos principais operadores Linq para filtrar e efetuar projeções no resultado. Você pode realizar uma combinação de operadores, em especial os operadores de filtro e projeção, utilizando uma sintaxe Linq do C#, e o EF se encarrega de converter sua *query* Linq em uma *query* SQL. Veja o exemplo da Figura 19.

Figura 19 – Exemplo de *Select* usando *querys* Linq no banco via *DbContext*

```
List<Autor> autoresComUrl = await dbContext.Autores
                                        .Where(a => a.WebUrl != null)
                                        .ToListAsync();

Console.WriteLine("Count autoresComUrl:" + autoresComUrl.Count);
//Output no console:
//Count autoresComUrl: 1

List<string> nomes = await dbContext.Autores.Select(a => a.Nome)
                                            .ToListAsync();

Console.WriteLine("Nomes retornados: " + nomes.Count);
foreach (var nome in nomes)
{
    Console.Write($"{nome},");
}
//Output no console:
//Nomes retornados: 2
//JON SMITH, JOSEPH ALBAHARI,
```

Assim como ocorre em *queries* Linq, no EF Core temos também a funcionalidade de execução tardia, ou *lazy evaluation*. Isso pode ajudar muito quando estamos construindo *queries* mais complexas ou nas quais possamos dinamicamente decidir entre trazer mais ou menos dados da agregação, como *joins* (veremos isso em detalhes no próximo tema). Logo, para a *query* realmente ser executada no banco de dados, será necessário que você a termine com algum dos seguintes operadores (todos possuem implementações síncronas e assíncronas):

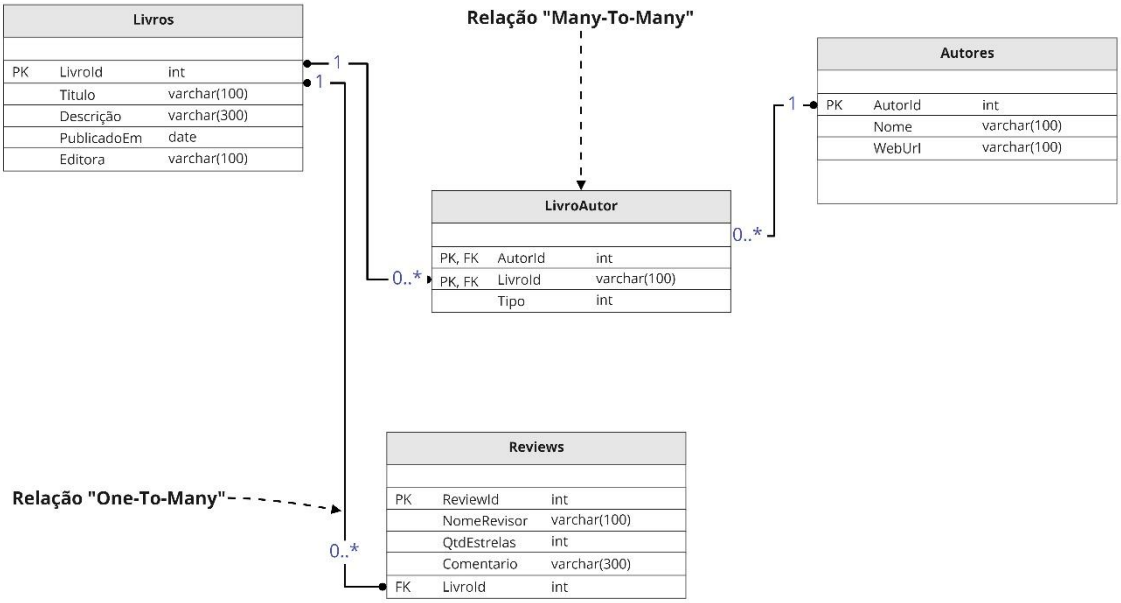
- *ToList*, *ToDictionary*, *ToLookup*, *ToArray*, *ToHashSet*;
- *First*, *FirstOrDefault*;
- *Last*, *LastOrDefault*;
- *Single*, *SingleOrDefault*;
- *Find*;
- *Count*, *Any*, *LongCount*, *Contains*.

TEMA 3 – RELAÇÕES NO EF

Vamos agora entender como podemos representar relações entre classes que o EF consiga mapear e nos ajudar em *queries* que naturalmente envolveriam *joins*, em uma consulta SQL. Primeiramente, retomaremos nosso modelo de dados, a que adicionaremos mais uma entidade. A seguir, criaremos uma entidade que represente a quantidade de *reviews* que determinado livro obteve.

Além disso, vamos permitir que um livro esteja vinculado a mais de um autor, discriminando se o autor é **coautor** ou **autor principal** do livro (Figura 20).

Figura 20 – Nova estrutura do banco: vinculação de um livro a mais de um autor



Nesse novo modelo, temos duas relações entre classes distintas:

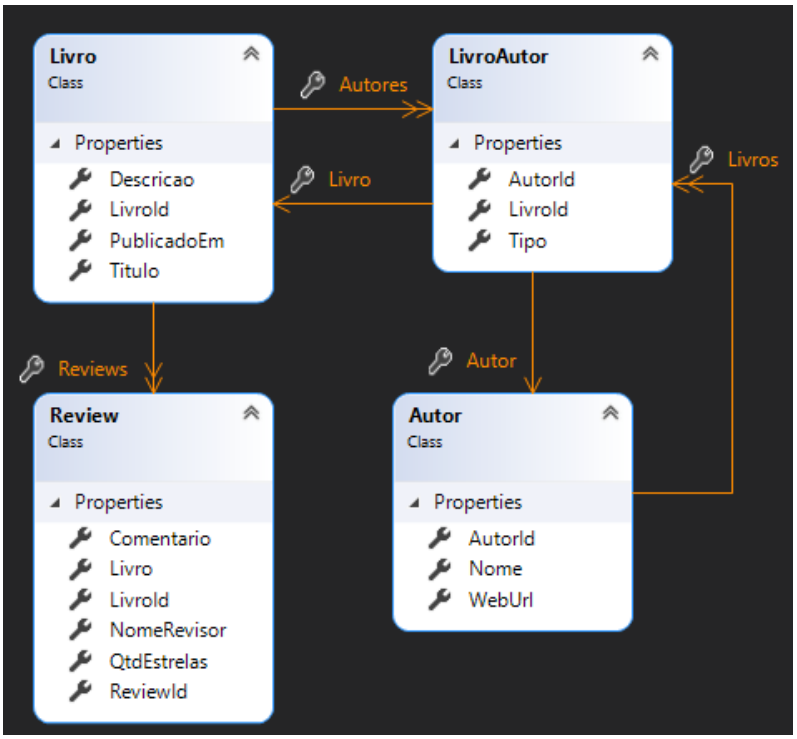
1. *One to many* (um para muitos) – utilizada para representar uma relação entre uma entidade que sempre existirá e outra que pode existir, mas sempre estará vinculada à primeira, sendo zero, uma ou mais instâncias.
2. *Many to many* (muitos para muitos) – utilizada para representar uma relação em que uma instância da entidade 1 pode se relacionar com mais de uma instância da entidade 2 e vice-versa.

Esse tipo de modelagem é comum quando olhamos para modelos relacionais. Mas, no mundo da OO, modelos *many to many* tendem a causar bastante discussão. Isso porque a sua representação, em OO, pode ser feita de forma mais simples, em que a entidade *Livro* contém uma lista de entidades *Autores* e a entidade *Autor*, uma lista de entidades *Livros*. O EF, na sua versão 5.0 em diante, suporta modelos *many to many* nativamente, sem a necessidade de uma classe (entidade) extra para modelar essa relação. Mas, em nosso caso, adicionamos um valor *Tipo* à relação. Dessa forma, seremos obrigados a ter uma classe que modele essa entidade *Autores + Livros para um Tipo*.

3.1 Configurando o *mapping* do EF para relações

O diagrama de classes da Figura 21 apresenta a configuração final das nossas classes (entidades) para refletir o novo modelo de banco proposto na Figura 20.

Figura 21 – Diagrama de classes do modelo final



Agora, vamos configurar o *mapping*, primeiro da *review* e depois do *LivroAutor* (Figura 22).

Figura 22 – Configuração do *mapping* da classe *Review*

```
public class ReviewMapp : IEntityTypeConfiguration<Review>
{
    3 references
    public void Configure(EntityTypeBuilder<Review> builder)
    {
        builder.HasKey(x => x.ReviewId);

        builder.Property(x => x.NomeRevisor)
               .HasMaxLength(100);

        builder.Property(x => x.QtdEstrelas);

        builder.Property(x => x.Comentario)
               .HasMaxLength(300);

        builder.HasOne(x => x.Livro)
               .WithMany(x=> x.Reviews)
               .HasForeignKey(x=> x.LivroId);
    }
}
```

1 review esta associada a 1 livro;
1 livro possui muitas reviews;
A FK da review que relaciona o livro é a LivroId;

```
public class LivroAutorMapp : IEntityTypeConfiguration<LivroAutor>   A relação many-to-many, precisa ser
{                                                                     configurada para ambas as "pontas".
    3 references
    public void Configure(EntityTypeBuilder<LivroAutor> builder)
    {
        builder.HasKey(x => new { x.LivroId, x.AutorId });   Configura uma PK dupla, usando as duas chaves;

        builder.HasOne(x => x.Livro)                         1 AutorLivro possui um Livro relacionado;
               .WithMany(x => x.Autores)                     Cada Livro possui muitos Autores;
               .HasForeignKey(x => x.LivroId);               A FK que relaciona eles é a "LivroId";

        builder.HasOne(x => x.Autor)                         1 AutorLivro possui um Autor relacionado;
               .WithMany(x => x.Livros)                      Cada Autor possui muitos Livros;
               .HasForeignKey(x => x.AutorId);               A FK que relaciona eles é a "AutorId";
    }
}
```

Agora que temos nossas relações devidamente mapeadas, podemos executar uma nova *migration*. Isso irá criar um *script* com todas as alterações necessárias, no banco, para refletir nosso novo modelo. Para isso, basta executar o comando de adicionar uma *migration* e, depois, o comando *Update database*, conforme fizemos no Tema 1.

TEMA 4 – *QUERY* EM MODELOS COMPLEXOS NO EF

Para realizar consultas em entidades com relações, vamos montar alguns cenários como:

a. listar todos os autores baseando-se nas *reviews* de seus livros, ordenando do autor mais bem avaliado para o menos bem avaliado;
b. listar os livros com as melhores *reviews* (de três a cinco estelas) e agrupá-los por ano de lançamento;
c. listar os autores com a maior quantidade de publicações, indicando quantas como autor principal e quantas como coautor;
d. *top* de três livros menos avaliados e seus comentários de avaliação (*reviews*).

Os exemplos detalhados podem ser encontrados no projeto desta aula. Recomendamos que você o execute e o acompanhe junto com a aula, para facilitar o entendimento e visualizar o que ocorre no código. Os dados do projeto de exemplo são gerados dinamicamente com funções randômicas; logo, os seus resultados não serão idênticos aos relatados nos exemplos de *query* a seguir. O importante é a estrutura da *query*.

Figura 23 – *Query* 1: autores com base nas *reviews*

```
var q1 = dbContext.Autores.AsNoTracking()
                  .Include(a => a.Livros)
                  .Select(a => new
                  {
                    autor = a,
                    qtdPontuacao = a.Livros.Sum(autorLivro =>
                                                autorLivro.Livro
                                                          .Reviews
                                                          .Sum(r => r.QtdEstrelas)
                                               )
                  })
                  .OrderByDescending(x=> x.qtdPontuacao);

foreach (var elemento in await q1.ToListAsync())
{
    Console.WriteLine($"Autor: {elemento.autor.Nome} |
                        Qtd Pontuacao: {elemento.qtdPontuacao}");
}
//Output no console:
//Autor: JON SMITH | Qtd Pontuacao: 1727
//Autor: JOSEPH ALBAHARI | Qtd Pontuacao: 807
//Autor: ALAN ARAYA | Qtd Pontuacao: 435
```

Logo nessa primeira *query*, podemos ver o uso do operador *Include*. Com ele, estamos informando, para o EF, que, além do DbSet de autores, nós desejamos levar para o resultado final, também, a relação de livros desses autores. A cláusula *Include* será transformada **em um *join*, pelo EF**. Muitas vezes, como é o caso de relações *many to many*, se você utilizar uma propriedade da outra ponta da relação, vamos dizer, de livros, o **EF também carregará os registros da tabela de livros para o resultado final**. O *Include* pode ser combinado com o operador *.ThenInclude()*, o qual pode formar uma **cascata** de inclusões. Você pode encontrar mais referências para essa situação no documento oficial presente em: <https://docs.microsoft.com/pt-br/ef/core/querying/related-data/eager> (Carregamento, 2020).

Observe também como fizemos o *Sum* de todas os pontos (estrelas) das *reviews* dos livros. Por último, organizamos o resultado de forma decrescente, utilizando o **tipo anônimo retornado pela projeção do *Select()***. Nessa projeção, criamos um novo **objeto** dinamicamente, com duas propriedades: ***Autor*, com o objeto *Autor* nele (variável *a*); e *QtdPontuacao*, com a soma dos pontos das estrelas de todas as *reviews* do livro**.

Figura 24 – Query 2: livros com melhores *reviews*, agrupados pelo seu ano de lançamento

```
var q2 = dbContext.Livros.AsNoTracking()
                  .Include(l => l.Reviews.Where(r => r.QtdEstrelas >= 3))
                     .ToListAsync();

//Como o MYSql não suporta o grupping de Data do EF
//Vamos agrupa-lo localmente baseado no resultado da query:
var livrosAgrupadosPorAno = q2.GroupBy(l => l.PublicadoEm.Year)
                              .OrderBy(g=> g.Key);

foreach (var elementoAgrupado in livrosAgrupadosPorAno)
{
    Console.WriteLine($"Ano: {elementoAgrupado.Key}");
    //Ordena baseado na média da pontuação, dos mais bem pontuados para os menos
    foreach (var livro in elementoAgrupado.OrderByDescending(l=> l.Reviews.Average(r=> r.QtdE
strelas)))
    {
        //Como a função Average retorna um double, arrendondamos ele para 2 digitos decimais
        Console.WriteLine($"    Média: {Math.Round(livro.Reviews.Average(r =>     r.QtdEstrel
as),2)} | Livro: {livro.Titulo} - {livro.Descricao}");
    }

}
//Output no console:
/*
Ano: 2005
    Média: 3,5 | Pontuacao: 105 | Livro: Java Fundamentos - Fundamentos de programaçao em Jav
a
Ano: 2016
    Média: 3,63 | Pontuacao: 98 | Livro: C# Fundamentos - Fundamentos de programaçao em C#
    Média: 3,33 | Pontuacao: 80 | Livro: Linguagens de programacao - Um olhar holistico para
linguagens de programacao
    Média: 3,53 | Pontuacao: 53 | Livro: Kotlin Fundamentos - Fundamentos de programaçao em K
otlin
    Média: 3,78 | Pontuacao: 34 | Livro: O Futuro da Quantica - Discussao metafisica de Fisic
a Quantica
Ano: 2017
    Média: 3,37 | Pontuacao: 128 | Livro: Xamarin Fundamentos - Fundamentos de Xamarin
    Média: 3,4 | Pontuacao: 17 | Livro: Windows Forms morreu! Salve o WPF - Visao sobre o nov
o conceito de dev para desktops
Ano: 2018
    Média: 3,71 | Pontuacao: 89 | Livro: Xamarin Zero-To-Hero - Zero to hero series
    Média: 3,38 | Pontuacao: 88 | Livro: Asp.Net com Kubernetes - K8s + AspNet
    Média: 3,43 | Pontuacao: 48 | Livro: Asp.Net Core Zero-To-
Hero - Asp.Net zero to hero lhe transforma em um programador web excelente
Ano: 2019
    Média: 3,52 | Pontuacao: 148 | Livro: Xamarin Rocks! - C# para mobile, melhor impossível!
    Média: 3,24 | Pontuacao: 55 | Livro: Xamarin e MUAI components - Xamarin MUAI - Multi-
platform App UI
    Média: 3,36 | Pontuacao: 47 | Livro: C# Rocks - C# é a melhor linguagem
Ano: 2020
    Média: 3,51 | Pontuacao: 123 | Livro: Xamarin navigations - Dominando a navegaçao em Apps
 Xamarin
Ano: 2021
    Média: 3,7 | Pontuacao: 37 | Livro: Asp.Net APIs - dominando as APIs em .NET
    Média: 3,5 | Pontuacao: 14 | Livro: Asp.Net Microservices - Microservices o futuro!
 */
```

Nessa *query* 2, podemos ver o uso de uma instrução nova no EF, presente em suas versões mais novas, com a qual podemos fazer um **filtro diretamente na cláusula *Include***. Isso pode ajudar muito quando não queremos “trazer” tudo da ponta do operador incluído. Nesse caso, estamos selecionando apenas *reviews* com três estrelas ou mais. A consulta leva todos os dados para a memória, onde, depois, realizamos um *GroupBy* pelo ano de publicação.

Infelizmente, o EF não suporta, nativamente, a função *Year*, do MySQL, não conseguindo fazer esse agrupamento de forma simples, no banco.

Após tê-los agrupados, percorremos os elementos. Nesse caso do item da lista *LivrosAgrupadosPorAno*, haverá um objeto **similar a uma chave/valor** de um *dictionary*, em que a chave é item agrupado, no caso o ano de lançamento do livro; e o valor é uma lista dos dados que contêm aquela chave em comum (ano de lançamento em comum). O operador *Average()* faz então o cálculo da média de pontos em memória, pois os dados já foram obtidos do banco. Desde o operador *GroupBy*, as ações são Linq em lista de objetos puros. A instrução *AsNoTracking()* também é importante. Com ela, estamos falando para o EF não controlar o estado de nossas entidades. Dessa forma, alterações nos objetos retornados não serão rastreadas pelo EF.

Figura 25 – *Query* 3: autores com a maior quantidade de livros, por tipo de autoria

```
var q3 = dbContext.Autores
                  .Include(a => a.Livros)
                  .Select(a => new
                  {
                     Publicacoes = a.Livros.Count,
                     PublicacoesComoAutorPrincial = a.Livros.Where(l=>           l
.Tipo == TipoAutor.Principal.ToString()).Count(),
                     PublicacoesComoCoautor = a.Livros.Where(l =>           l.Tipo
 == TipoAutor.Coautor.ToString()).Count(),
                     Nome = a.Nome
                   })
                   .OrderByDescending(x=> x.Publicacoes);

foreach (var item in await q3.ToListAsync())
{
    Console.WriteLine($"Autor: {item.Nome} | QtdPublicacoes: {item.Publicacoes} |
Como principal: {item.PublicacoesComoAutorPrincial} | Como CoAutor: {item.Publicac
oesComoCoautor}");
}
//Output no console
//Autor: JON SMITH | QtdPublicacoes: 16 | Como principal: 16 | Como CoAutor: 0
//Autor: JOSEPH ALBAHARI | QtdPublicacoes: 7 | Como principal: 0 | Como CoAutor: 7
//Autor: ALAN ARAYA | QtdPublicacoes: 4 | Como principal: 0 | Como CoAutor: 4
```

A *query* 3 é uma *query* relativamente simples, mas que demonstra o poder do uso do operador *Select*. Nela, podemos ver exemplos subquerys utilizando o mesmo objeto para se contar a quantidade de publicações e a quantidade de publicações como autor e coautor. Outro fator interessante é sua ordenação de forma decrescente por um caso gerado dentro do *Select*, o campo *Publicacoes*. Esse campo não existe no objeto *Autor*, pois é um campo (propriedade) que geramos dentro de um objeto anônimo, no operador *Select*. Nele, contabilizamos a quantidade de livros total em *Autor*.

Figura 26 – *Query* 4: *top* três livros menos avaliados e seus comentários

```
//Top 3 livros menos avaliados e seus comentários de avalição
var q4 = dbContext.Livros
                  .Include(l => l.Reviews)
                  .Select(l => new
                  {
                    Livro = l.Titulo,
                    QtdPontuacao = l.Reviews.Sum(r => r.QtdEstrelas),
                    ReviewsBaixa = l.Reviews.Where(r => r.QtdEstrelas < 3)
                  })
                  .OrderBy(x => x.QtdPontuacao)
                  .Take(3);


foreach (var item in await q4.ToListAsync())
{
    Console.WriteLine($"Livro: {item.Livro} | Pontuação: {item.QtdPontuacao}");
    Console.WriteLine($"    Reviews:");
    foreach (var review in item.ReviewsBaixa)
    {
        Console.WriteLine($"    -Comentário:{review.Comentario} -
Revisor: {review.NomeRevisor}");
    }
}
//Output no console:
/*
Livro: Asp.Net Microservices | Pontuaçao: 16
    Reviews:
    -Comentário:Achei fraco! - Revisor: ALAN
    -Comentário:Achei fraco! - Revisor: ALAN
Livro: Windows Forms morreu! Salve o WPF | Pontuaçao: 21
    Reviews:
    -Comentário:Mais ou menos - Revisor: GABRIEL
    -Comentário:Mais ou menos - Revisor: JOAO
Livro: O Futuro da Quantica | Pontuaçao: 52
    Reviews:
    -Comentário:Achei fraco! - Revisor: SIMONE
    -Comentário:Mais ou menos - Revisor: ISABELA
    -Comentário:Mais ou menos - Revisor: ISABELA
    -Comentário:Achei fraco! - Revisor: PAULO
    -Comentário:Mais ou menos - Revisor: GABRIELA
    -Comentário:Mais ou menos - Revisor: SIMONE
    -Comentário:Achei fraco! - Revisor: JOSIANE
    -Comentário:Mais ou menos - Revisor: MATHEUS
    -Comentário:Mais ou menos - Revisor: GABRIELA
    -Comentário:Mais ou menos - Revisor: JOSIANE
    -Comentário:Achei fraco! - Revisor: MARIA
*/
```

Nessa *query* 4, combinamos o uso de projeção com filtros. Quando um dos itens da projeção fornece um valor discreto (um valor único), isso irá gerar o equivalente a subconsultas, no banco de dados. Quando coletamos uma lista de valores, como no caso das *reviews* de notas baixas, isso irá gerar um *join* comum, pois o EF precisa do conjunto de elementos e não apenas de um valor discreto para gerar essa *output*. Por fim, temos o uso de mais um operador, o *Take()*, com o qual podemos especificar um **limite** de dados que serão retornados. Isso seria equivalente ao *top* ou *limit* das instruções em SQL.

O EF não é o único *framework* disponível e utilizado no mercado, como já falamos, mas é o mais usado pela comunidade de desenvolvedores .NET, sem dúvida. No entanto, quando se trata de situações reais, você poderá se deparar com sistemas mais antigos ou que possuem necessidades específicas, a que um ORM como o EF não atenda plenamente.

Uma alternativa aos ORMs que mapeiam nossas entidades e criam o seu rastreamento etc. é o uso de bibliotecas que nos permitam nos aproximar mais do banco de dados e do modo nativo de trabalho do .NET, como no caso do ADO.NET. O ADO.NET é um conjunto de funcionalidades do .NET que permite o acesso a bancos de dados, em especial aos relacionais. Nasceu praticamente junto com o C# e o .NET e foi muito empregado até a versão Core do .NET. Após a versão Core ter chegado ao mercado, muitas camadas do ADO foram reescritas e permitiram entrada de novas bibliotecas e inclusive o próprio avanço e evolução do EF (Smith, 2018, p. 58).

Uma das bibliotecas mais populares entre os desenvolvedores .NET e que aproxima o desenvolvedor de uma experiência mais próxima do ADO, ao mesmo tempo que oferece uma série de novidades e melhorias para o tratamento de dados, é o Dapper. O Dapper é uma biblioteca conhecida como micro-ORM, isso pois ela não dispõe de todas as funcionalidades de um ORM como o EF ou o NHibernate, mas há nela muitas funções úteis e fáceis de utilizar, além do mapeamento de uma *query* para objetos, assim como o EF faz. O Dapper é um projeto de *open source* disponível no GitHub e gratuito como o EF Core. O Dapper tem o seu foco em *performance* e é um excelente aliado para quem precisa de operações extremamente performáticas ou precisa escrever *queries* SQL diretamente no código, para tirar o máximo proveito do banco de dados utilizado (o EF também permite isso, mas é menos performático que o Dapper, nesse ponto, além de ser mais pesado, com muito mais *dynamic link libraries* – DLLs e códigos associados a seus projetos). O Dapper, contudo, não suporta o conceito de *code first* como o EF; logo, ele não gera *migrations*; e nem o *change tracking* da forma como o EF faz.

Nos próximos tópicos, vamos utilizar o Dapper para fazer algumas operações no modelo de dados que já estamos trabalhando. Então, focaremos

apenas nos pontos em que mudam as implementações do EF para o Dapper. Para instalar o Dapper, basta fazer *download* do pacote NuGet Dapper.

5.1 Entendendo o Dapper

O Dapper precisa de pouquíssima ou nenhuma configuração inicial para ser utilizado. Com o *namespace* do Dapper no *using* de sua classe, você já pode dar início às chamadas para o banco de dados.

O Dapper gera uma série de métodos de extensão dentro do objeto *ConnectionString*. Esses métodos podem ser empregados para chamadas do banco de dados como as que veremos a seguir. Muitas dessas chamadas são extensões para consulta, mas algumas também possibilitam a chamada de operações **unitárias** ou **discretas**. As principais extensões para utilização no Dapper e seu comportamento consistem em:

- ***Execute*/*ExecuteAsync*** – executa um comando no banco de dados e retorna com a quantidade de linhas afetadas. Pode ser um *insert*, um *update* ou um *delete*, por exemplo.
- ***ExecuteScalar*/*ExecuteScalarAsync*** – executa um comando no banco de dados e retorna com um **valor único**. Pode ser, por exemplo, a ID gerada pelo *insert* de uma *identity column*.
- ***Query*/*QueryAsync*** – executa um comando no banco de dados e retorna com o *IEnumerable<T>* de um objeto. Pode retornar um *type* dinâmico (*dynamic*) ou mesmo um *type* específico passado como parâmetro no *<T>*. O Dapper irá mapear os nomes das colunas retornadas na *query* com os nomes das propriedades de sua classe e carregar o seu objeto com os seus respectivos valores. O retorno será sempre com um *IEnumerable<T>* e não um *List<T>*. Para converter o resultado em lista, deve-se primeiro executar a *query* e, depois, o comando *ToList()* de *IEnumerable*, para realizar a conversão.
- ***QueryMultiple*/*QueryMultipleAsync*** – executa mais de um comando simultaneamente, dentro da mesma *query*, por exemplo, dois *Selects* ou uma *StoreProcedure* que produza mais de um *result set* de retorno, podendo assim mapear o resultado de cada objeto distinto fazendo uso de uma entidade chamada *GridReader*.

Como mostra o trecho de código da Figura 27, para utilizar o Dapper basta abrir uma conexão com o banco de dados. Com base em uma *DbConnection*, nesse nosso caso, uma *MySqlConnection*, você já pode usar os métodos de extensão do Dapper. Como ele se encontra em uma classe *DbConnection*, que é uma classe base para todas as conexões com bancos de dados relacionais no C#, em tese o Dapper pode se **adaptar** a qualquer banco de dados: MySQL, SQL Server, Oracle Database, PostgreSQL, SQLite, Firebird etc.

Figura 27 – Exemplo de como inicializar uma conexão e um comando de *Select* no banco

```
using (var dbConnection = new MySqlConnector.MySqlConnection(stringDeConexao))
{
    string query = @"SQL COMMAND";

    return  await dbConnection.QueryAsync<T>(query);
}
```

5.2 Operações Crud com o Dapper

Vamos mostrar agora os comandos Crud de nossa classe *Autor* e como eles podem ser executados via Dapper.

Figura 28 – Exemplo de *Insert* da entidade *Autor* com Dapper (similar ao *Create* no EF)

```
public async Task<bool> InserirAutorAsync(Autor autor)
{
    try
    {
        using (var dbConnection = new MySqlConnector.MySqlConnection(stringDeConexao))
        {
            string query = @"INSERT INTO Autores
                        VALUES(@nome, @weburl);";

            await dbConnection.ExecuteAsync(query, autor);
        }
        return true;
    }
    catch (Exception ex)
    {
        Console.WriteLine("Erro ao inserir o autor. Erro: " + ex.Message);
        return false;
    }
}
```

Observe como é simples o uso do Dapper. Basta escrevermos uma *query* SQL, como, no caso, um comando *Insert*, e passá-lo ao *ExecuteAsync*. Para mapear os parâmetros da *query*, a melhor forma é utilizar parâmetros dinâmicos,

como no exemplo da Figura 28, em que *@nome* e *@weburl* irão gerar parâmetros para o banco MySQL. O Dapper irá procurar em nosso objeto *Autor*, passado para o método *execute*, propriedades com o mesmo nome e, com base nelas, gerar parâmetros específicos ao banco de dados. Essa forma é a recomendada para se transmitir parâmetros ao Dapper, utilizando @ na *query*, pois assim você evitará dar *replace* da *string* e mandar seu parâmetro como um texto livre, correndo risco de *SQL injection*. É justamente devido à dificuldade de passagem correta de parâmetros que várias linguagens e bibliotecas de acesso a bancos de dados possuíam que tantos sites e sistemas ficaram expostos, durante anos, aos ataques de *SQL injection*.

Figura 29 – Exemplo de *Update* da entidade *Autor* no Dapper

```
public async Task<bool> UpdateAutorAsync(Autor autor)
{
    try
    {
        using (var dbConnection = new MySqlConnector.MySqlConnection(strin
gDeConexao))
        {
            string query = @"UPDATE Autores
                             SET Nome = @nome,
                                 WebUrl = @weburl
                             WHERE AutorId = @autorid";

            await dbConnection.ExecuteAsync(query, autor);
        }
        return true;
    }
    catch (Exception ex)
    {
        Console.WriteLine("Erro ao atualizar o autor. Erro: " + ex.Message
);
        return false;
    }
}
```

O comando *Update* pode ser executado com o mesmo método *ExecuteAsync* que utilizamos no *Insert*, pois não temos linhas para retornar em nossa *query* SQL. Observe a necessidade de protegermos nosso método com um *Try/Catch*. Isso porque podem ocorrer muitas exceções nesse pequeno trecho de código, desde abertura da conexão a até mesmo um erro no banco de dados, na execução de nossa *query*. Nesse caso, o Dapper irá lançar uma exceção com o erro. Diferentemente do EF, que pré-executa validações quando realizamos os comandos *Add* ou *Remove*, o Dapper manipula diretamente o

banco de dados e não trabalha com o conceito de DbSet, tampouco *change tracking* de entidades retornadas.

Figura 30 – Exemplo de *Remove*/*Delete* da entidade *Autor* no Dapper

```
using (var dbConnection = new MySqlConnector.MySqlConnection(stringDeConex
ao))
{
    string query = @"DELETE FROM Autores
                     WHERE AutorId = @autorid";

    await dbConnection.ExecuteAsync(query, autorId);
}
```

Nesse exemplo da Figura 30, encurtamos o código *boilerplate* (código clichê) em torno do método de *Delete* e deixamos apenas a parte importante. Note a similaridade dessa operação com os comandos *Update* e *Insert*.

Figura 31 – *Read*/*Query* no Dapper

```
public async Task<List<Autor>> ListaTodosAutores()
{
    try
    {
        using (var dbConnection = new MySqlConnector.MySqlConnection(strin
gDeConexao))
        {
            string query = @"SELECT * FROM Autores";

            var resultado = await dbConnection.QueryAsync<Autor>(query);

            return resultado.ToList();
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine("Erro ao consultar os dados no BD. Erro: " + ex.
Message);
        return new List<Autor>();
    }
}
```

Observe, agora, o uso do método *Query<T>*. Com ele, o Dapper pode executar um comando SQL e, ainda, retornar um *result set*, convertendo-o em um tipo de objetos específicos, no caso, um objeto *Autor*. No exemplo da Figura 31, não transmitimos parâmetro algum. Vamos fazê-lo com as próximas consultas.

5.3 *Queries* complexas com o Dapper

Vamos agora replicar duas das *queries* mais complexas que fizemos utilizando EF, no Dapper. Para isso, a parte mais trabalhosa será escrever a *query* SQL propriamente dita, mas teremos que lidar com um problema diferente no Dapper: até o momento, não tínhamos utilizado, no EF, os chamados *objetos flat*.

O Dapper irá retornar com um *result set* no banco de dados. O que significa que ele não irá preencher objetos com hierarquias complexas automaticamente. Apesar de isso ser possível com emprego de algumas configurações e do pacote *Dapper.FluentMap*, vamos nos ater ao retorno *flat* e a como utilizá-lo, nesse momento, pois pode nos ser muito útil, no dia a dia, trabalhar com objetos que chamamos de *data transfer objects* (DTOs), ou seja, objetos que não nos têm serventia para comportamentos (métodos) e servem apenas para que possamos representar nossos dados em objetos. É muito comum usarmos DTOs para representar relatórios e outras visões, sobre os dados de um sistema, que não correspondam a nenhuma estrutura de entidades, pois contêm dados sumarizados, agregados ou misturados; e, para isso, precisamos de uma representação do resultado da *query* no banco de dados.

Figura 32 – *Query* complexa, no Dapper

```
string query = @"SELECT
                        r2.ReviewId,
                           r2.NomeRevisor,
                           r2.QtdEstrelas,
                           r2.Comentario,
                           l1.*
                   FROM
                (
                SELECT
                   l.LivroId,
                               l.Titulo,
                               (
                      SELECT
                       SUM(r.QtdEstrelas)
                      FROM Review as r
                      WHERE
                        r.LivroId = l.LivroId
                               ) as 'QtdPontuacao'
                      FROM
                      Livros AS l
                     ORDER BY QtdPontuacao ASC
                     LIMIT 3
                        ) as l1
                       LEFT JOIN
                       (
                        SELECT *
                              FROM Review as rv
                        WHERE  rv.QtdEstrelas <= 3
                       ) as r2 on l1.LivroId = r2.LivroId;";

var resultado = await dbConnection.QueryAsync<LivroBaixaPontuacaoDTO>(query);
```

Novamente, removemos o código *boilerplate* da *query* da Figura 32 para focarmos apenas na parte mais importante. Observe a complexidade da estrutura da *query*. É claro que existem algumas outras maneiras de resolvermos esse *Select* no MySQL, mas essa é uma das mais performáticas. O objeto *LivroBaixaPontuacaoDTO* precisa ser criado para representar os campos retornados pela nossa *query*, que mistura dados de duas entidades e ainda possui um campo computado (*QtdPontuacao*). Para isso, a forma mais simples é criarmos um objeto DTO e mapearmos o retorno da *query* nesse objeto. Na Figura 33 podemos ver a estrutura do DTO criado para representar o retorno de nossa *query*. Note que ele não possui hierarquia ou relacionamentos, é apenas um objeto com propriedades soltas. Chamamos isso de *objeto flat*.

Figura 33 – Exemplo de DTO criado no Dapper para retorno da *query*

```
namespace ConsoleApp.Aula.DTO
{
    public class LivroBaixaPontuacaoDTO
    {
        public int LivroId { get; set; }
        public string Titulo { get; set; }
        public int QtdPontuacao { get; set; }
        public int ReviewId { get; set; }
        public string NomeRevisor { get; set; }
        public int QtdEstrelas { get; set; }
        public string Comentario { get; set; }
    }
}
```

```
string query = @"   SELECT l2.LivroId,
                           l2.Titulo,
                           l2.Descricao,
                           YEAR(l2.PublicadoEm) as 'AnoPublicacao',
                           SUM(r.QtdEstrelas) as 'Pontuacao',
                           AVG(r.QtdEstrelas) as 'Media'
                    FROM Livros l2
                    LEFT JOIN Review r ON r.LivroId = l2.LivroId
                    WHERE r.QtdEstrelas >= 3
                    GROUP BY l2.LivroId,
                                    l2.Titulo,
                                    l2.Descricao,
                                    YEAR(l2.PublicadoEm)
                    ORDER BY Media DESC;";

var resultado = await dbConnection.QueryAsync<LivroPontuacaoPorAnoDTO>(que
ry);
```

No modelo de *query* do segundo trecho de código da Figura 33, foi possível resolver todo o seu agrupamento e processamento no banco de dados sem necessariamente trazer mais dados para resolver isso no *client-side*,

utilizando Linq nos objetos em memória. É claro que, para facilitar a montagem do resultado final no *console*, usamos Linq e o código será muito similar ao código Linq, quando utilizado o EF. A questão toda está no processamento dos dados. Quando o EF não pode resolver sua consulta e você está lidando com um grande volume de dados (milhares ou milhões de linhas), a *tunning* da *query* para o tipo de banco que você está utilizando pode fazer toda a diferença. E, nesse quesito, o Dapper alia muito bem facilidade de uso com elaboração de consultas mais personalizadas, no banco de dados, permitindo ainda o mapeamento do resultado para uma lista de objetos.

Na Figura 34, podemos ver o objeto *flat* que criamos para representar o retorno da *query* disposta no segundo trecho de código da Figura 33.

Figura 34 – Exemplo de DTO *flat* criado para representar retorno da *query*

```
public class LivroPontuacaoPorAnoDTO
{
    public int LivroId { get; set; }
    public string Titulo { get; set; }
    public string Descricao { get; set; }
    public int AnoPublicacao { get; set; }
    public int Pontuacao { get; set; }
    public decimal Media { get; set; }
}
```

FINALIZANDO

Nossa aula de interações com o banco de dados no C# chega ao fim, mas apenas arranhamos a ponta do *iceberg*, no que diz respeito aos *frameworks* e possibilidades oferecidas por eles. O EF é um *framework* de ORM muito poderoso e fácil de utilizar. Como vimos nesta aula, ele é capaz de combinar o melhor que temos a oferecer em termos de rastreamento e controle de mudanças nas entidades com consultas nas quais o desenvolvedor não precisa dominar o SQL e sim apenas a sintaxe Linq e o funcionamento das *queries* no EF. Isso o torna um *framework* **multibanco de dados**, diminuindo muito a necessidade do desenvolvedor de alterar seu código para que ele funcione como um banco de dados diferente. Todos as *queries* que elaboramos nesta aula funcionarão perfeitamente em SQL Server ou PostgreSQL, sem que seja necessário alterar nenhuma linha de código.

Pudemos também aprender sobre o poder do “micro-ORM” Dapper. Mesmo que sua *query* careça de uma adaptação ou outra para cada banco de

dados que sua aplicação utilizar, é possível trabalhar com o Dapper criando algumas camadas de abstração. Ele pode ser um grande aliado para aplicações que necessitem de extrema performance ou que precisem tirar o máximo proveito do banco de dados usado, trabalhando com funções e recursos específicos ao *provider* de banco diretamente nas queries SQL executadas pelo Dapper. O Dapper ainda possui uma série de extensões em outros pacotes NuGets, que podem oferecer desde comandos Cruds automáticos (de modo similar ao que o EF faz) até mapeamentos mais complexos de objetos hierárquicos.

Novamente, recomendamos que você execute o código de exemplo desta aula. Também recomendamos fortemente que você explore as referências citadas, como uma forma de se aprofundar mais nos assuntos discutidos e ter ainda mais domínio dos recursos possíveis no EF e no Dapper.

REFERÊNCIAS

CARREGAMENTO adiantado de dados relacionados. **Microsoft Docs**, 8 set. 2020. Disponível em: <https://docs.microsoft.com/pt-br/ef/core/querying/related-data/eager>. Acesso em: 8 out. 2021.

CONTROLE de alterações em EF Core. **Microsoft Docs**, 30 dez. 2020. Disponível em: <https://docs.microsoft.com/pt-br/ef/core/change-tracking/>. Acesso em: 8 out. 2021.

ENTITY Framework Core. **Entity Framework Tutorial**, [20--]. Disponível em: <https://www.entityframeworktutorial.net/efcore/entity-framework-core.aspx>. Acesso em: 8 out. 2021.

INÍCIO rápido: instalar e usar um pacote no Visual Studio (somente Windows). **Microsoft Docs**, 24 jul. 2018. Disponível em: <https://docs.microsoft.com/pt-br/nuget/quickstart/install-and-use-a-package-in-visual-studio>. Acesso em: 8 out. 2021.

REFERÊNCIA de ferramentas de Entity Framework Core-CLI do .NET Core. **Microsoft Docs**, 27 out. 2020. Disponível em: <https://docs.microsoft.com/pt-br/ef/core/cli/dotnet>. Acesso em: 8 out. 2021.

SMITH, J. **Entity Framework Core in Action**. 1. ed. Shelter Island: Manning Publications Co., 2018.

VIDA útil, configuração e inicialização do DbContext. **Microsoft Docs**, 7 nov. 2020. Disponível em: <https://docs.microsoft.com/pt-br/nuget/quickstart/install-and-use-a-package-in-visual-studio>. Acesso em: 8 out. 2021.

VISÃO geral das migrações. **Microsoft Docs**, 28 out. 2020. Disponível em: <https://docs.microsoft.com/pt-br/nuget/quickstart/install-and-use-a-package-in-visual-studio>. Acesso em: 8 out. 2021.