Entity Framework Core

Pre-requisitos

- Docker
- .NET SDK 5
- IDE (e.g: Visual Studio 2019 Community Edition, Visual Studio Code o JetBrains Rider)
- Cliente UI de SqlServer (e.g: Visual Studio SqlServer Explorer, Azure Data Studio o SQL Server Management Studio)

Introducción

Vamos a hacer un breve repaso teórico de lo que son las bases de datos relacionales (i.e: RDBMS), del lenguaje SQL y del problema de la adaptación de impedancias objeto-relacional, para llegar a los ORM como Entity Framework Core y explorar su uso y ventajas.

Ejemplo - sqlclient-soccer

Para este ejemplo necesitamos un servidor de base de datos SqlServer en localhost (ver Anexo 1) y un cliente UI de SqlServer (ver Anexo 2).

En el cliente UI de SqlServer, conecta a Microsoft SQL Server en `localhost` con usuario `sa` y la contraseña especificada en la variable de entorno del contenedor docker (e.g: `Lem0nCode!`) y explora las bases de datos.

Clona el repositorio con el ejemplo

```
git clone https://gitlab.com/sunnyatticsoftware/lemoncode/sqlclient-soccer.git
```

y abre la solución, que contiene un proyecto de consola .NET 5 con una dependencia a la librería `Microsoft.Data.SqlClient`.

```
public class SoccerClient
{
    private readonly string _databaseName;
    private readonly SqlConnection _sqlConnection;

    public SoccerClient(string connectionString, string databaseName)
    {
        _databaseName = databaseName;
        _sqlConnection = new SqlConnection(connectionString);
    }

    public Task Connect()
    {
        return _sqlConnection.OpenAsync();
    }
```

```
    public Task DeleteDatabase()
    {
        var sqlDropDatabase = $"DROP DATABASE IF EXISTS {_databaseName};";
        var sqlDropDatabaseCommand = new SqlCommand(sqlDropDatabase,
_sqlConnection);
        return sqlDropDatabaseCommand.ExecuteNonQueryAsync();
    }

    public async Task CreateDatabase()
    {
        var sqlCreateDatabase = $"CREATE DATABASE {_databaseName};";
        var sqlCreateDatabaseCommand = new SqlCommand(sqlCreateDatabase,
_sqlConnection);
        await sqlCreateDatabaseCommand.ExecuteNonQueryAsync();

        var sqlUseDatabase = $"USE {_databaseName};";
        var sqlUseDatabaseCommand = new SqlCommand(sqlUseDatabase,
_sqlConnection);
        sqlUseDatabaseCommand.ExecuteNonQuery();
    }

    public async Task CreateTables()
    {
        var sqlCreateTableTeam =
            "CREATE TABLE Team ("
            + "Id INT NOT NULL,"
            + "Code CHAR(3) NOT NULL,"
            + "PRIMARY KEY(Id));";
        var sqlCreateTableOneCommand = new SqlCommand(sqlCreateTableTeam,
_sqlConnection);
        await sqlCreateTableOneCommand.ExecuteNonQueryAsync();

        var sqlCreateTableGame =
            "CREATE TABLE Game ("
            + "Id INT NOT NULL,"
            + "HomeTeamId INT NOT NULL,"
            + "AwayTeamId INT NOT NULL,"
            + "ScheduledOn DATETIME NOT NULL,"
            + "StartedOn DATETIME NULL,"
            + "EndedOn DATETIME NULL,"
            + "PRIMARY KEY(Id),"
            + "FOREIGN KEY (HomeTeamId) REFERENCES Team(Id),"
            + "FOREIGN KEY (AwayTeamId) REFERENCES Team(Id));";
        var sqlCreateTableTwoCommand = new SqlCommand(sqlCreateTableGame,
_sqlConnection);
        await sqlCreateTableTwoCommand.ExecuteNonQueryAsync();

        var sqlCreateTableGoal =
            "CREATE TABLE Goal ("
            + "Id INT NOT NULL,"
            + "TeamId INT NOT NULL,"
            + "ScoredBy VARCHAR(100) NOT NULL,"
            + "ScoredOn DATETIME NOT NULL,"
            + "PRIMARY KEY(Id),"
```

```
            + "FOREIGN KEY (TeamId) REFERENCES Team(Id));";
        var sqlCreateTableJoinCommand = new SqlCommand(sqlCreateTableGoal,
_sqlConnection);
        await sqlCreateTableJoinCommand.ExecuteNonQueryAsync();
    }

    public async Task InsertData()
    {
        var sqlInsertTeams =
            "INSERT INTO Team([Id],[Code]) VALUES"
            + "(1, 'RMA'),"
            + "(2, 'BAR'),"
            + "(3, 'MGA'),"
            + "(4, 'ATM'),"
            + "(5, 'ALV'),"
            + "(6, 'PAL'),"
            + "(7, 'SEV'),"
            + "(8, 'DEP'),"
            + "(9, 'CEL'),"
            + "(10, 'VAD'),"
            + "(11, 'GRA'),"
            + "(12, 'OSA'),"
            + "(13, 'RSO'),"
            + "(14, 'RAC'),"
            + "(15, 'COR'),"
            + "(16, 'EIB'),"
            + "(17, 'RAY'),"
            + "(18, 'VAL'),"
            + "(20, 'VIL'),"
            + "(19, 'ESP');";
        var sqlInsertTeamsCommand = new SqlCommand(sqlInsertTeams,
_sqlConnection);
        await sqlInsertTeamsCommand.ExecuteNonQueryAsync();

        var sqlInsertGames =
            "INSERT INTO Game([Id],[HomeTeamId],[AwayTeamId],[ScheduledOn],
[StartedOn],[EndedOn]) VALUES"
            + "(1, 1, 2, '2021-10-27 18:00:00.000', '2021-10-27 18:00:10.100',
null),"
            + "(2, 3, 4, '2021-10-27 18:00:00.000', '2021-10-27 18:00:20.200',
null),"
            + "(3, 5, 6, '2021-10-27 20:00:00.000', null, null),"
            + "(4, 7, 8, '2021-10-28 18:00:00.000', null, null),"
            + "(5, 9, 10, '2021-10-28 20:00:00.000', null, null);";
        var sqlInsertGamesCommand = new SqlCommand(sqlInsertGames,
_sqlConnection);
        await sqlInsertGamesCommand.ExecuteNonQueryAsync();

        var sqlInsertGoals =
            "INSERT INTO Goal([Id],[TeamId],[ScoredBy],[ScoredOn]) VALUES"
            + "(1, 1, 'Karim Benzema', '2021-10-27 18:05:25.000'),"
            + "(2, 1, 'Vinicius Jr', '2021-10-27 18:07:10.000'),"
            + "(3, 3, 'Antonio Cortés', '2021-10-27 18:07:15.000'),"
            + "(4, 4, 'Luis Suarez', '2021-10-27 18:10:00.200');";
```

```
        var sqlInsertGoalsCommand = new SqlCommand(sqlInsertGoals,
_sqlConnection);
        await sqlInsertGoalsCommand.ExecuteNonQueryAsync();
    }

    public async Task QueryData(Action<string> linePrinter)
    {
        var sqlQueryTeams =
            "SELECT t.Code "
            + "FROM Team t;";
        var sqlQueryTeamsCommand = new SqlCommand(sqlQueryTeams, _sqlConnection);
        var teamsReader = await sqlQueryTeamsCommand.ExecuteReaderAsync();
        linePrinter.Invoke("\nTeams:");
        while (await teamsReader.ReadAsync())
        {
            var code = teamsReader.GetString(0);
            linePrinter.Invoke(code);
        }
        teamsReader.Close();

        var sqlQueryGames =
            "SELECT th.Code, ta.Code, g.StartedOn "
            + "FROM Game AS g "
            + "JOIN Team AS th On g.HomeTeamId = th.Id "
            + "JOIN Team AS ta On g.AwayTeamId = ta.Id "
            + "WHERE g.StartedOn IS NOT null;";
        var sqlQueryGamesCommand = new SqlCommand(sqlQueryGames, _sqlConnection);
        var gamesReader = await sqlQueryGamesCommand.ExecuteReaderAsync();
        linePrinter.Invoke("\nGames:");
        while (await gamesReader.ReadAsync())
        {
            var teamHomeCode = gamesReader.GetString(0);
            var teamAwayCode = gamesReader.GetString(1);
            var startedOn = gamesReader.GetDateTime(2);
            linePrinter.Invoke($"{teamHomeCode} - {teamAwayCode} started on
{startedOn}");
        }
        gamesReader.Close();
    }

    public async Task Disconnect()
    {
        await _sqlConnection.CloseAsync();
        await _sqlConnection.DisposeAsync();
    }
}
```

La funcionalidad se usa con una pequeña clase `Program.cs` que crea una instancia pasándole el connection string y el nombre de la base de datos.

```
class Program
{
    static async Task Main(string[] args)
    {
        try
        {
            await RunExample();
        }
        catch (Exception exception)
        {
            Console.WriteLine(exception);
        }

        Console.WriteLine("Press [ENTER] to end");
        Console.ReadLine();
    }

    private static async Task RunExample()
    {
        const string connectionString = "Data Source=localhost;User
Id=sa;Password=Lem0nCode!;";
        const string databaseName = "sqlclient_soccer";
        var soccerClient = new SoccerClient(connectionString, databaseName);
        await soccerClient.Connect();

        await soccerClient.DeleteDatabase();
        await soccerClient.CreateDatabase();
        Console.WriteLine($"Database {databaseName} re-created");

        await soccerClient.CreateTables();
        Console.WriteLine("Tables created");

        await soccerClient.InsertData();
        Console.WriteLine("Data inserted");

        await soccerClient.QueryData(Console.WriteLine); // Same as x =>
Console.WriteLine(x)

        await soccerClient.Disconnect();
    }
}
```

¿Podrían los alumnos implementar funcionalidad adicional? Por ejemplo, se puede añadir lo siguiente:

- Una consulta que obtenga todos los goles marcados en la jornada
- Una consulta que obtenga el listado de los goleadores junto con el número de goles
- Una consulta que obtenga para cada partido todos los goles anotados
- etc.

¿Qué incomodidades se detectan al trabajar de esta forma con bases de datos? ¿Cuál sería el proceso a seguir si quisiéramos añadir para cada gol una descripción que registre cómo se marcó? En todos los goles ya

registrados en la base de datos ¿qué mostraría la columna de descripción?

Ejemplo - efcore-soccer

Los ORM nos facilitan la vida cuando trabajamos con bases de datos relacionales. En este ejemplo vamos a ver cómo podemos transformar esas tablas en entidades con las que operar como si fueran simples objetos C#.

También vamos a ver paso a paso cómo utilizar la CLI de EF Core para crear bases de datos y aplicar migraciones cada vez que queramos cambiar el esquema de la base de datos.

Creamos un nuevo proyecto `LemonCode.EfCore.Soccer`. Dentro del proyecto, en un nuevo directorio al que podemos llamar `Entities`, creamos las clases (i.e: entidades) que describen cada una de las tablas.

Para equipos

```
public class TeamEntity
{
    public int Id { get; set; }
    public string Code { get; set; }

    // Navigation properties
    public List<GoalEntity> Goals { get; set; } = new();
}
```

Para partidos

```
public class GameEntity
{
    public int Id { get; set; }
    public int HomeTeamId { get; set; }
    public int AwayTeamId { get; set; }
    public DateTime ScheduledOn { get; set; }
    public DateTime? StartedOn { get; set; }
    public DateTime? EndedOn { get; set; }

    // Navigation properties
    public TeamEntity HomeTeam { get; set; }
    public TeamEntity AwayTeam { get; set; }
    public List<GoalEntity> Goals { get; set; } = new();
}
```

y para goles

```
public class GoalEntity
{
    public int Id { get; set; }
    public int TeamId { get; set; }
```

```
    public int GameId { get; set; }
    public string ScoredBy { get; set; }
    public DateTime ScoredOn { get; set; }

    // Navigation properties
    public TeamEntity Team { get; set; }
    public GameEntity Game { get; set; }
}
```

Cabe destacar que estas entidades no tienen ninguna dependencia a Entity Framework ni a ninguna otra librería. Son simples POCO (Plain Old CLR Object), modelos anémicos para guardar estado, no comportamiento (aunque pueden tener funciones si así se desea).

Ahora necesitamos el `DbContext` donde se describen los `DbSet` que identifican a las tablas como colección de entidades. Añade primero una dependencia nuget `Microsoft.EntityFrameworkCore.SqlServer` al proyecto para tener acceso al `DbContext`.

```
public class SoccerContext
    : DbContext
{
    private const string ConnectionString =
"Server=localhost;Database=efcore_soccer;user=sa;password=Lem0nCode!";

    public DbSet<TeamEntity> Teams { get; set; }
    public DbSet<GameEntity> Games { get; set; }
    public DbSet<GoalEntity> Goals { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        => optionsBuilder.UseSqlServer(ConnectionString);

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder
            .Entity<GameEntity>()
            .HasOne<TeamEntity>(x => x.HomeTeam)
            .WithMany()
            .OnDelete(DeleteBehavior.Restrict);

        modelBuilder
            .Entity<GameEntity>()
            .HasOne<TeamEntity>(x => x.AwayTeam)
            .WithMany()
            .OnDelete(DeleteBehavior.Restrict);

    }
}
```

El connection string describe la cadena de conexión y el nombre de la base de datos a crear/utilizar. Dado que el modelo entidad-relación concreto que estamos utilizando especifica que un `Game` tiene dos referencias a `Team`, una para el equipo local y otra para el visitante, hay que configurar explícitamente esa particularidad, ya

que Entity Framework es capaz de inferir relaciones entre entidades pero, en estos casos, debemos ser explícitos en cuanto al comportamiento al eliminar o actualizar tablas.

Es muy importante observar como los DbSet son enumerables `IQueryable` y, por lo tanto, soportan LINQ.

Una vez creadas las entidades y el `DbContext`, podemos utilizar la CLI de Entity Framework Core para generar la base de datos en modo *code first*

Instala el CLI de Entity Framework Core con

```
dotnet tool install --global dotnet-ef
```

o con

```
dotnet tool update --global dotnet-ef
```

para tener la última versión.

Para crear una migración, o la inicial cuando se desea crear la base de datos, ejecuta desde el directorio raíz de la solución:

```
dotnet ef migrations add inicial --project src/LemonCode.EfCore.Soccer
```

Un directorio llamado `Migrations` será automáticamente creado.

Para aplicar las migraciones que hayan sido generadas, ejecuta:

```
dotnet ef database update --project src/LemonCode.EfCore.Soccer
```

Ya hemos sido capaces de crear la base de datos siguiendo un *code first* y utilizando la herramienta EF Core CLI y migraciones.

Recapitulando, si especificamos qué entidades queremos tener, qué propiedades tienen y cómo se relacionan entre ellas, Entity Framework es capaz de generar automáticamente todo el código SQL necesario para crear bases de datos, tablas, claves primarias, claves foráneas e incluso índices.

Pero una cosa es crear ese código SQL en forma de migraciones, y otra muy diferente ejecutarlo. Por eso hay dos pasos: crear la migración y actualizar la base de datos.

Si observamos la base de datos podemos ver una tabla especial llamada `dbo._EFMigrationsHistory` que lleva la cuenta de qué migraciones se han aplicado en la base de datos. Por lo tanto, si queremos saber en qué version se encuentra el esquema de nuestra base de datos, esta tabla nos dará una pista.

Ya estamos listos para programar asumiendo que hay una base de datos y utilizando Entify Framework para operar con ella sin necesidad de introducir ni una sola línea de lenguaje SQL. Simplemente, como si tuviésemos colecciones de objetos disponibles y todo el potencial de LINQ.

Vamos a crear un `Program.cs` con código para instanciar nuestro `DbContext` y para insertar datos.

```
class Program
{
    static async Task Main(string[] args)
    {
        try
        {
            await RunExample();
        }
        catch (Exception exception)
        {
            Console.WriteLine(exception);
        }

        Console.WriteLine("Press [ENTER] to end");
        Console.ReadLine();
    }

    private static async Task RunExample()
    {
        var realMadrid = new TeamEntity { Code = "RMA" };
        var barcelona = new TeamEntity { Code = "BAR" };
        var malaga = new TeamEntity { Code = "MGA" };
        var atletico = new TeamEntity { Code = "ATM" };

        var teams =
            new List<TeamEntity>
            {
                realMadrid,
                barcelona,
                malaga,
                atletico,
                new() { Code = "ALV" },
                new() { Code = "PAL" },
                new() { Code = "SEV" },
                new() { Code = "DEP" },
                new() { Code = "CEL" },
                new() { Code = "VAD" },
                new() { Code = "GRA" },
                new() { Code = "OSA" },
                new() { Code = "RSO" },
                new() { Code = "RAC" },
                new() { Code = "COR" },
                new() { Code = "EIB" },
                new() { Code = "RAY" },
                new() { Code = "VAL" },
                new() { Code = "VIL" },
```

```
                new() { Code = "ESP" }
            };

        var games =
            new List<GameEntity>
            {
                new()
                {
                    HomeTeam = teams.Single(x => x.Code == "RMA"),
                    AwayTeam = teams.Single(x => x.Code == "BAR"),
                    ScheduledOn = new DateTime(2021, 10, 27, 18, 0, 0),
                    StartedOn = new DateTime(2021, 10, 27, 18, 0, 10),
                    EndedOn = null,
                    Goals =
                        new List<GoalEntity>
                        {
                            new()
                            {
                                ScoredBy = "Karim Benzema",
                                Team = realMadrid,
                                ScoredOn = new DateTime(2021, 10, 27, 18, 5, 25)
                            },
                            new()
                            {
                                ScoredBy = "Vinicius Jr",
                                Team = realMadrid,
                                ScoredOn = new DateTime(2021, 10, 27, 18, 5, 25)
                            }
                        }
                },
                new()
                {
                    HomeTeam = teams.Single(x => x.Code == "MGA"),
                    AwayTeam = teams.Single(x => x.Code == "ATM"),
                    ScheduledOn = new DateTime(2021, 10, 27, 18, 0, 0),
                    StartedOn = new DateTime(2021, 10, 27, 18, 0, 10),
                    EndedOn = null,
                    Goals =
                        new List<GoalEntity>
                        {
                            new()
                            {
                                ScoredBy = "Antonio Cortés",
                                Team = malaga,
                                ScoredOn = new DateTime(2021, 10, 27, 18, 7, 15)
                            },
                            new()
                            {
                                ScoredBy = "Luis Suarez",
                                Team = atletico,
                                ScoredOn = new DateTime(2021, 10, 27, 18, 10, 00)
                            }
                        }
                }
```

```
            };

        await using var soccerContext = new SoccerContext();
        await soccerContext.Games.AddRangeAsync(games);
        // Implicitly, goals and teams are added
        await soccerContext.SaveChangesAsync();
        Console.WriteLine("Data inserted");
    }
}
```

Es muy importante observar lo que hemos hecho. No hemos creado explícitamente todas y cada una de las tablas. Simplemente hemos hecho una inserción en el `DbSet` de los partidos (i.e: `DbSet<GameEntity>`) de objetos. Esos objetos tienen propiedades que son referencias a otros objetos, como equipos y goles. Entity Framework es capaz de inferir todo lo que necesita insertar en la base de datos a partir de nuestros objetos instanciados en memoria y añadidos a uno o varios `DbSet`. Cuando le indicamos al `DbContext` que guarde los cambios, Entity Framework enviará todo el código SQL a la base de datos.

Hemos conseguido abstraernos totalmente del proveedor de base de datos (nos da igual si usamos SqlServer, o MySql, o PostgreSQL), y del lenguaje SQL. Solamente necesitamos conocer que estamos utilizando un paradigma de persistencia relacional. De lo demás se encarga Entity Framework.

Ahora vamos a ver cómo podemos leer datos de la base de datos, es decir, cómo hacer queries con Entity Framework.

Extraemos toda la funcionalidad de inserción de datos en una función o clase diferente y comentamos la llamada a esa función, para no volver a ejecutar esa parte.

Creamos una nueva función para leer datos.

Primero mostramos todos los equipos de la tabla `Team` y podemos ver lo sencillo que resulta iterar sobre simples objetos dentro de enumerables `DataSet`

```
await using var soccerContext = new SoccerContext();

// Get teams
var teams = await soccerContext.Teams.ToListAsync();
Console.WriteLine("Teams:");
foreach (var team in teams)
{
    Console.WriteLine(team.Code);
}
```

También podemos filtrar resultados en base de datos

```
// Get teams filtered in DB
var teamsFiltered = await soccerContext.Teams.Where(x =>
x.Code.Contains("M")).ToListAsync();
Console.WriteLine("\nTeams containing 'M':");
```

```
foreach (var team in teamsFiltered)
{
    Console.WriteLine(team.Code);
}
```

Compara la anterior consulta con esta.

```
// Get teams filtered in memory
var teamsFilteredInMemory = (await soccerContext.Teams.ToListAsync()).Where(x =>
x.Code.Contains("M"));
Console.WriteLine("\nTeams containing 'M':");
foreach (var team in teamsFilteredInMemory)
{
    Console.WriteLine(team.Code);
}
```

¿Qué diferencias hay?

Aparentemente no hay diferencias, en ambos casos veo el resultado esperado en consola, que son todos los equipos que contienen una letra 'M' en su código. Pero para Entity Framework no es lo mismo, como vamos a ver.

Visual Studio tiene un `Performance Profiler` (e.g: Debug > Performance Profiler o combinación de teclas ALT + F2) que podemos utilizar par aanalizar lo que ocurre en nuestro proceso.

Seleccionamos la casilla `Database` para examinar queries y analizar cuánto tiempo tardan. Si la aplicación se está ejecutando en modo `Release` dará resultados más realistas que en modo `Debug`.

Selecciona las casillas deseadas (e.g: .NET Async, CPU Usage, Database) y pulsa el botón Start. La aplicación arrancará y veremos un link para parar la recolección de datos llamado `Stop collection` que generará un reporte en el que se pueden apreciar las queries SQL lanzadas por Entity Framework.

En el caso anterior, se puede apreciar que la primera consulta es transformada por Entity Framework en

```
SELECT [t].[Id], [t].[Code] FROM [Teams] AS [t] WHERE [t].[Code] LIKE N'%M%'
```

En la siguiente consulta vemos que lo que se envía a la base de datos es en realidad

```
SELECT [t].[Id], [t].[Code] FROM [Teams] AS [t]
```

Es decir, hay una gran diferencia entre filtrar **antes** de hace run `.ToListAsync()` o hacerlo después. Si se hace antes, el filtrado ocurre en base de datos y es mucho más eficiente (i.e: más rápido porque no hace falta que todos los datos de la BD viajen por la red, solamente el resultado final). Si se hace después, el filtrado estará ocurriendo en memoria.

Veamos esto de la ejecución diferida con más detalle. Si una query no se envía a la base de datos hasta que se utiliza, explícitamente, un método terminador (e.g: `ToList()`, `First()`, `FirstOrDefault()`, `Last()`, `LastOrDefault()`, `Single()`), entonces en el `Performance Profiler` no deberíamos ver ningna query llegando a la base de datos, como se puede comprobar con

```
// Get teams query without hitting database
var teamsQuery = soccerContext.Teams.Where(x =>
x.Code.Contains("M")).AsQueryable();
var teamsQueryWithAdditionalFilter = teamsQuery.Where(x =>
x.Code.Contains("A")).AsQueryable();
var sqlQuery = teamsQueryWithAdditionalFilter.ToQueryString();
Console.WriteLine($"Query without hitting database: {sqlQuery}");
```

Sin embargo, podemos apreciar cómo la consulta anterior es simplemente una query que no llega a lanzarse pero, al ser un `queryable`, podemos ir componiendo filtros dinámicamente, lo cual es muy útil para filtrado, paginación, sorting, etc. La consulta anterior produce el siguiente código SQL que nunca llega a ejecutarse en la base de datos.

```
Query without hitting database:
DECLARE @__p_0 int = 1;
DECLARE @__p_1 int = 2;

SELECT [t].[Id], [t].[Code]
FROM [Teams] AS [t]
WHERE ([t].[Code] LIKE N'%M%') AND ([t].[Code] LIKE N'%A%')
ORDER BY (SELECT 1)
OFFSET @__p_0 ROWS FETCH NEXT @__p_1 ROWS ONLY
```

Si quisiéramos ejecutar la consulta anterior deberíamos añadir un

```
var teamsDelayedExecution = await
teamsQueryWithAdditionalFilterAndPagination.ToListAsync();
Console.WriteLine("\nTeams delayed execution, skipping 1:");
foreach (var team in teamsDelayedExecution)
{
    Console.WriteLine(team.Code);
}
```

Como comentario adicional, otro aspecto muy importante del `Performance Profiler` es que en la vista de Async Activities podemos ver qué funciones asíncronas se han esperado y completado y cuáles no. Por ejemplo, haz una llamada a una función asíncrona sin esperar la respuesta y finaliza la aplicación inmediatamente para ver cómo quedan ejecuciones asíncronas sin finalizar.

```
var goalsTask = soccerContext.Goals.ToListAsync(); // no tiene await
Environment.Exit(0);
```

Ahora vamos a explorar las diferencias entre eager loading y lazy loading. Primero vamos a ver qué es lo que pasa cuando hacemos lo siguiente.

```
var teamsWithGoalsAttempted = await soccerContext.Teams.ToListAsync();
Console.WriteLine("Teams (with goals?):");
foreach (var team in teamsWithGoalsAttempted)
{
    Console.WriteLine(team.Code);
    var goals = team.Goals;
    foreach (var goal in goals)
    {
        Console.WriteLine($"{goal.ScoredOn} - {goal.ScoredBy}");
    }
}
```

Aunque podríamos pensar que seremos capaces de ver cada equipo con los goles anotados, el resultado es que no se han recuperado goles para cada uno de los equipos. Si analizamos el profiler vemos que la consulta SQL que se envía a la base de datos es

```
SELECT [t].[Id], [t].[Code] FROM [Teams] AS [t]
```

Sin rastro, por lo tanto, de la tabla goles. ¿Por qué? ¿Cómo podemos conseguirlo?

Una opción sería usando *eager loading* indicando que queremos incluir resultados de la tabla satélite. Por ejemplo:

```
// Eager loading
var teamsWithGoalsEagerLoading = await soccerContext.Teams.Include(x =>
x.Goals).ToListAsync();
Console.WriteLine("\nTeams with goals (eager loading):");
foreach (var team in teamsWithGoalsEagerLoading)
{
    Console.WriteLine(team.Code);
    var goals = team.Goals;
    foreach (var goal in goals)
    {
        Console.WriteLine($"{goal.ScoredOn} - {goal.ScoredBy}");
    }
}
```

Ahora sí, gracias al `.Include(x => x.Goals)` podemos ver los goles que cada equipo tiene en su cuenta particular. Si analizamos el profiler vemos que la query SQL ejecutada en base de datos es:

```
SELECT [t].[Id], [t].[Code], [g].[Id], [g].[GameId], [g].[ScoredBy], [g].
[ScoredOn], [g].[TeamId]
FROM [Teams] AS [t] LEFT JOIN [Goals] AS [g] ON [t].[Id] = [g].[TeamId]
ORDER BY [t].[Id], [g].[Id]
```

Es decir, gracias a incluir la tabla satélite podemos ver que hay un join en base de datos.

Hay otra alternativa, el lazy loading, por la cual podemos cargar los datos que necesitamos justo en el momento en el que los necesitemos, pero no antes. Para ello necesitamos hacer un par de cambios

- Añadir una dependencia nuget al proyecto `Microsoft.EntityFrameworkCore.Proxies`
- Indicar al proveedor de base de datos de Entity Framework que deseamos utilizar lazy loading. En el `DbContext`:

  ```
  protected override void OnConfiguring(DbContextOptionsBuilder
  optionsBuilder)
        => optionsBuilder
            .UseLazyLoadingProxies() // Esta línea se añade
            .UseSqlServer(ConnectionString);
  ```

- Hacer **todas** las propiedades de navegación de las entidades como virtuales. Si no, tendremos un error en tiempo de ejecución

  ```
  public class GameEntity
  {
    public int Id { get; set; }
    public int HomeTeamId { get; set; }
    public int AwayTeamId { get; set; }
    public DateTime ScheduledOn { get; set; }
    public DateTime? StartedOn { get; set; }
    public DateTime? EndedOn { get; set; }

    // Navigation properties
    public virtual TeamEntity HomeTeam { get; set; }
    public virtual TeamEntity AwayTeam { get; set; }
    public virtual List<GoalEntity> Goals { get; set; } = new();
  }

  public class GoalEntity
  {
    public int Id { get; set; }
    public int TeamId { get; set; }
    public int GameId { get; set; }
    public string ScoredBy { get; set; }
    public DateTime ScoredOn { get; set; }

    // Navigation properties
  ```

```
    public virtual TeamEntity Team { get; set; }
    public virtual GameEntity Game { get; set; }
  }

  public class TeamEntity
  {
    public int Id { get; set; }
    public string Code { get; set; }

    // Navigation properties
    public virtual List<GoalEntity> Goals { get; set; } = new();
  }
```

Ahora estamos listos para lazy loading y podemos hacer lo siguiente, sin el `Include()`, para ver todos los goles debajo de cada equipo:

```
// Lazy loading (tras añadir UseLazyLoadingProxies() y hacer las propiedades de
navegación virtuales)
var teamsWithGoalsLazyLoading = await soccerContext.Teams.ToListAsync();
Console.WriteLine("\nTeams with goals (lazy loading):");
foreach (var team in teamsWithGoalsLazyLoading)
{
    Console.WriteLine(team.Code);
    var goals = team.Goals;
    foreach (var goal in goals)
    {
        Console.WriteLine($"{goal.ScoredOn} - {goal.ScoredBy}");
    }
}
```

Si analizamos el profiler veremos algo muy interesante. Ya no solamente hay una consulta ejecutada en la base de datos. Ahora hay muchas.

```
SELECT [t].[Id], [t].[Code] FROM [Teams] AS [t]

SELECT [g].[Id], [g].[GameId], [g].[ScoredBy], [g].[ScoredOn], [g].[TeamId]
FROM [Goals] AS [g]
WHERE [g].[TeamId] = @__p_0

SELECT [g].[Id], [g].[GameId], [g].[ScoredBy], [g].[ScoredOn], [g].[TeamId]
FROM [Goals] AS [g]
WHERE [g].[TeamId] = @__p_0

SELECT [g].[Id], [g].[GameId], [g].[ScoredBy], [g].[ScoredOn], [g].[TeamId]
FROM [Goals] AS [g]
WHERE [g].[TeamId] = @__p_0

SELECT [g].[Id], [g].[GameId], [g].[ScoredBy], [g].[ScoredOn], [g].[TeamId]
```

```
FROM [Goals] AS [g]
WHERE [g].[TeamId] = @__p_0
```

Se puede observar que hay una primera consulta para extraer información de equipos. Y, después, para cada uno de los equipos encontrados, Entity Framework compone una misma consulta, aunque parametrizada, y la ejecuta una y otra vez en cada paso de bucle.

Desde luego, no hay nada más vago que hacer consultas solamente cuando se necesite. Esto, que puede parecer ineficiente, puede tener cabida en algunos escenarios, como en aquellos donde querramos cargar los goles de un equipo solamente cuando exploremos ese equipo en concreto, pero no queremos traer toda la información en un primer momento porque puede ser demasiada. ¿Se nos ocurre alguno más?

Como programadores, debemos tener los pros y contras muy presentes a la hora de diseñar nuestras consultas en el código.

Como nota adicional, cuando un contexto ha incluido una tabla satélite, si se reutiliza ese contexto, los datos que tuviera de esa tabla satélite se aprovecharán en futuras consultas.

Finalmente, falta por ver mejor el potencial de las migraciones que ya hemos utilizado para crear la base de datos en un primer momento.

Supongamos ahora que la base de datos se ha creado con *code_first* y que, por tanto, existe una migración en la base de datos.

Vamos a añadir un cambio en la estructura de la base de datos. Por ejemplo, una nueva propiedad `Description` en la entidad `GoalEntity`

```
public class GoalEntity
{
    public int Id { get; set; }
    public int TeamId { get; set; }
    public int GameId { get; set; }
    public string ScoredBy { get; set; }
    public DateTime ScoredOn { get; set; }
    public string Description { get; set; } // nueva propiedad

    // Navigation properties
    public TeamEntity Team { get; set; }
    public GameEntity Game { get; set; }
}
```

En una consola ejecutamos lo siguiente para generar una migración

```
dotnet ef migrations add gol_tiene_descripcion --project
src/LemonCode.EfCore.Soccer
```

y observamos la nueva clase generada. Se aprecia que hay dos partes fundamentales `Up`, que corresponde a los cambios y `Down` que corresponde al *rollback* de la operación si esta falla.

La base de datos se puede actualizar con la nueva migración con

```
dotnet ef database update --project src/LemonCode.EfCore.Soccer
```

Anexo 1 - SqlServer con docker (Anexo)

Crea un contenedor de SqlServer para tener un servidor de base de datos corriendo en local

```
docker run -e "ACCEPT_EULA=Y" \
-e "SA_PASSWORD=Lem0nCode!" \
-e "MSSQL_PID=Express" \
-p 1433:1433 \
--name sqlserver \
-d mcr.microsoft.com/mssql/server:2017-latest-ubuntu
```

Asegúrate de que el contenedor extá en ejecución con

```
docker ps
```

Si ya tenías ese contenedor (u otro), puedes simplemente arrancarlo

```
docker start sqlserver
```

Anexo 2 - Azure Data Studio

Descarga e instala Azure Data Studio para tu sistema operativo https://docs.microsoft.com/en-us/sql/azure-data-studio/download-azure-data-studio?view=sql-server-ver15

Anexo 3 - Eliminar base de datos de Azure Data Studio

Azure Data Studio no permite la eliminación de una base de datos desde la interfaz gráfica, pero se puede hacer

```
use master;
go
ALTER DATABASE mi_base_de_datos SET SINGLE_USER WITH ROLLBACK IMMEDIATE;
```

```
go
drop database mi_base_de_datos;
```