

# Appendix C: New Features in Entity Framework Core 2.1

This online appendix introduces the new, and in some cases improved, features in version 2.1 of the Entity Framework Core. We have included it as an online document because the first preview version of Entity Framework Core 2.1 had just been announced at the time our book was going into production in April of 2018. This appendix will be updated for the RTM version, which is currently planned for the summer of 2018.

## Group By

Listing C-1 shows a LINQ query with grouping using the LINQ grouping operator [group by or GroupBy()]. Although this query provides the desired result (number of flights per departure), the query takes a lot of time for large amounts of data. When researching causes, one quickly comes across the fact that the SQL command sent to the database management system completely lacks the grouping: Entity Framework Core has loaded all records and grouped them in the RAM, which is a bad and unexpected behavior.

**CAUTION:** In fact, Entity Framework Core version 1.x and 2.0 do not support translating a LINQ grouping into the GROUP BY syntax of SQL, which is a very frightening gap in Entity Framework Core. An improvement is planned only for version 2.1 of Entity Framework Core.

*Listing C-1. Determine the number of flights per place of departure.*

```
// Determine the number of flights per place of departure
using (var ctx = new WWWingsContext())
{
    var groups = from p in ctx.FlightSet
                  orderby p.FreePlaces
                  group p by p.Departure into g
                  select new { place = g.Key, number = g.Count ()};

    Console.WriteLine("number:" + groups.Count ());

    // Output
    foreach (var g in groups.ToList ())
    {
        Console.WriteLine (g.Place + ": " + g.Number);
    }
}
```

Listing C-1 sends the following SQL command to the database management system [twice, once for Count (), and once for ToList ()]:

```
SELECT [p].[FlightNo], [p].[Departure], [p].[Strikebound], [p].[CopilotId],
[p].[FlightDate], [p].[Airline], [p].[AircraftTypeNo], [p].[Free],
[p].[LastChange], [p].[Memo], [p].[Non-SmokerFlight], [p].[PilotId],
[p].[Places], [p].[Price], [p].[Timestamp], [p].[Destination]
FROM [Flight] AS [p]
ORDER BY [p].[Departure]
```

Unfortunately, in the current Preview 1 of Entity Framework Core 2.1 (Figure C-1), the grouping query is still unsuccessful. Entity Framework Core generates invalid SQL:

```
SELECT [p].[Departure], COUNT(*)
FROM [Flight] AS [p]
WHERE COUNT (*) > 5
GROUP BY [p].[Departure]
```

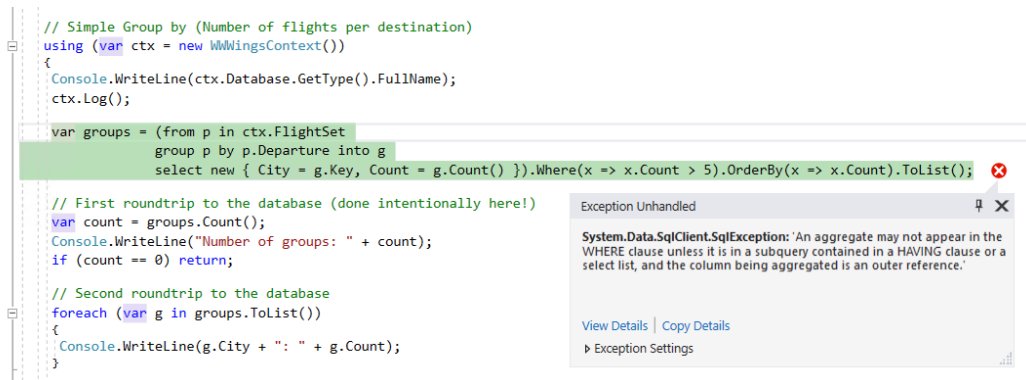


Figure C-1. Invalid SQL generation in Entity Framework Core 2.1 Preview 1

The error seems to be due to the count operator. I have reported this as a bug to Microsoft: <https://github.com/aspnet/EntityFrameworkCore/issues/11322>.

Other aggregate operators such as Min (), Max (), Sum (), and Average () work. This LINQ query:

```
var groups1 = (from p in ctx.FlightSet
               group p by p.Departure into g
               select new { City = g.Key, Min = g.Min(x => x.FreeSeats), Max = g.Max(x => x.FreeSeats), Sum = g.Sum(x => x.FreeSeats), Avg = g.Average(x => x.FreeSeats) }) ;
```

leads to this correct SQL in Entity Framework Core 2.1 Preview 1:

```
SELECT [p].[Departure], MIN ([p].[FreeSeats]), MAX ([p].[FreeSeats]),
SUM([p].[FreeSeats]), AVG (CAST([p].[FreeSeats] AS float))
FROM [Flight] AS [p]
GROUP BY [p].[Departure]
```

# Lazy Loading

While in the previous example Entity Framework Core just loads the flight and doesn't do anything else, the classic Entity Framework after the flight in Listing C-2 via lazy loading would load fully automatically the pilot and copilot information (with their other flights) and the bookings along with the passenger data. The entity framework would send a large number of SELECT commands to the database one after the other. How many there are depends on the number of passengers on this flight.

Microsoft has reintroduced automatic lazy loading into Entity Framework Core 2.1 as an option. It is not active in the standard.

## Activation of Lazy Loading

Unlike the classic Entity Framework, the developer in Entity Framework Core must explicitly enable lazy loading:

- In the project with the context class (here in the book: "DA"), the Nuget package newly introduced in the Entity Framework Core Version 2.1 is `Microsoft.EntityFrameworkCore.Proxies` and is to be installed:

```
install-package Microsoft.EntityFrameworkCore.proxies -pre
```

- In the context class in `OnConfiguring()` you have to add `UseLazyLoadingProxies()` before specifying the database driver:  
`Builder.UseLazyLoadingProxies().UseSqlServer(Connection String);`
- All navigation properties must be virtual. Listing C-3 shows this for the class "Flight".

*Listing C-3. Flight class with navigation properties declared virtual*

```
using System;

using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using EFCExtensions;

namespace BO
{

    [Serializable]
    public class Flight
    {
        ...

        #region Related Objects
```

```

public virtual Airline Airline { get; set; }
public virtual ICollection<Booking> BookingSet { get; set; }
public virtual Pilot Pilot { get; set; }
public virtual Pilot Copilot { get; set; }
[ForeignKey("AircraftTypeID")]
public virtual AircraftType AircraftType { get; set; }

// Explicit foreign key properties for the navigation properties
public string AirlineCode { get; set; } // mandatory!
public int PilotId { get; set; } // mandatory!
public int? CopilotId { get; set; } // optional
public byte? AircraftTypeID { get; set; } // optional
#endregion
}
}

```

With these changes, Listing C-3 for the flight also provides the pilots and co-pilots (each with their number of flights) and the passenger list (see Figure C-2).

### Demo LazyLoading

```

Flight Nr 101 from Berlin to Seattle has 122 free seats!
Pilot: Lafontaine has 1 flights as pilot!
Copilot: Özdemir has 3 flights as copilot!
Number of passengers on this flight: 8
Passengers on this flight:
- Passenger #6: Niklas Schröder
- Passenger #21: Julia Wolf
- Passenger #26: Leonie Fischer
- Passenger #46: Lukas Richter
- Passenger #49: Laura Schwarz
- Passenger #67: Lisa Weber
- Passenger #78: Anna Wolf
- Passenger #82: Leonie Müller

```

Figure C-2. Output of Listing C-3 with lazy loading

## Dangers of Lazy Loading

A multitude of SQL commands have now been sent to the database management system. Entity Framework Profiler reveals that there were 14 SQL commands (Figure C-3):

- First, the requested flight was loaded.
- Then the pilot was loaded from the employee table.
- Subsequently, all flights of this pilot were loaded from the flight table. (This is inefficient, because here alone the number of flights was needed. This happens whether you use the Property Count or the Count () LINQ extension method.)
- Now the co-pilot has been loaded from the employee table.
- Subsequently, all flights of this co-pilot were loaded from the flight table (which is inefficient, because here only the number of flights was needed. This happens whether you use the Property Count or the Count () LINQ extension method.)
- Then all bookings for this flight were downloaded from "Booking".
- Finally, for each passenger, the passenger information was loaded individually from the "Passenger" table.

Had it been more passengers, the number of SQL commands would have increased accordingly.

**WARNING:** Using lazy loading may seem convenient at first glance, but there is a danger that you will send a lot of SQL commands to the database management system without realizing it. I have often been called by customers to performance tuning missions, where the wrong use of lazy loading turned out to be a core problem. Lazy loading makes no sense if you know in advance that all records are needed (as in the example). Here you should use eager loading.

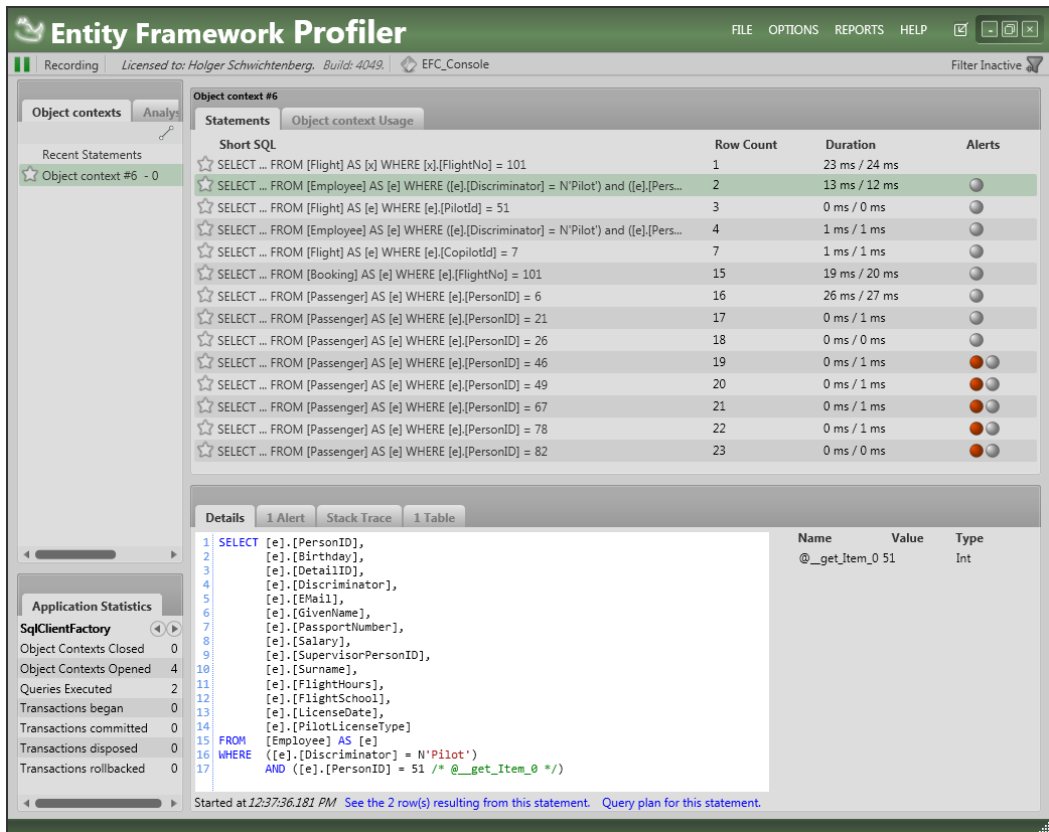


Figure C-3. 14 SQL commands have been sent.

**TIP:** Lazy loading makes sense for a master detail view on the screen. If there are many master records, it would be wasted time to load the detail records for each master record. Rather, one will always only request the detail records for the master record that the user has just clicked on. While at this point in the previous Entity Framework Core you could realize the master detail display via lazy loading without further program code when clicking on the master data set, you have to intercept the click in Entity Framework Core 2.1 and explicitly load the detail data records.

## Lazy loading without proxy classes

Lazy loading involves a special implementation challenge because the OR Mapper must catch any access to all object references in order to reload the connected objects as needed. The call `UseLazyLoadingProxies()` ensures that so-called runtime proxies are placed around all instances of the entity classes and Entity Framework Core thus intercepts all calls to navigation properties and checks whether the corresponding data has already been loaded.

Entity Framework Core also supports lazy loading without these runtime proxies. In this case, the package `Microsoft.EntityFrameworkCore.Proxies` and the call `UseLazyLoadingProxies()` are not necessary, but the implementation of the entity class and the navigation properties have to be extended:

- The entity class must be prepared for an `ILazyLoader` interface or `ILazyLoader.Load()` to be injected as a delegate in the constructor. The latter is preferable since `ILazyLoader` is in the `Microsoft.EntityFrameworkCore.dll`, which would force dependency of the GO project on Entity Framework Core entity classes.
- In the navigation properties, `Load()` must then be called.

The following listing shows a modification of the class `Passenger` with lazy loading without Runtime Proxies with injection of `ILazyLoader.Load()` as a delegate in the constructor of the class and calling this delegate in the getter of the navigation property of `BookingSet`.

*Listing C-4. Lazy loading without Runtime Proxies with Injection by `ILazyLoader.Load()` as a delegate*

```
using System;

using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Runtime.CompilerServices;

namespace BO
{
    public static class PocoLoadingExtensions
    {
        public static TRelated Load<TRelated>(
            this Action<object, string> loader,
            object entity,
            ref TRelated navigationField,
            [CallerMemberName] string navigationName = null)
            where TRelated : class
        {
            loader?.Invoke(entity, navigationName);

            return navigationField;
        }
    }

    public class PassengerStatus
    {
        public const char A = 'A';
    }
}
```

```

public const char B = 'B';
public const char C = 'C';

public static char[] PassengerStatusSet = { PassengerStatus.A,
PassengerStatus.B, PassengerStatus.C };
}

[Serializable]
public partial class Passenger : Person
{
    private Action<object, string> LazyLoader { get; set; }
    public Passenger()
    {
        this.BookingSet = new List<Booking>();
    }

    /// <summary>
    /// Injection for Lazy Loading without Runtime Proxies
    /// </summary>
    private Passenger(Action<object, string> lazyLoader)
    {
        LazyLoader = lazyLoader;
    }

    // Primary key is inherited!
    #region Primitive Properties
    public virtual Nullable<DateTime> CustomerSince { get; set; }

    public string FrequentFlyer { get; set; }

    [StringLength(1), MinLength(1), RegularExpression("[ABC]")]
    public virtual char Status { get; set; }
    #endregion

    #region Relations
    //public virtual ICollection<Booking> BookingSet { get; set; }

    private ICollection<Booking> _BookingSet;
    public ICollection<Booking> BookingSet
    {

```



```

// Lazy Loading without Runtime Proxies
get => LazyLoader?.Load(this, ref _BookingSet);
set => _BookingSet = value;
}
#endregion
}
}

```

A test client for this case shows Listing C-5, which loads a passenger and outputs all the bookings of this passenger.

*Listing C-5. Test of lazy loading without Runtime Proxy*

```

public static void Demo_LazyLoading_NoProxies()
{
    CUI.MainHeadline(nameof(Demo_LazyLoading_NoProxies));

    using (var ctx = new WWWingsContext())
    {
        ctx.Log();

        var p = ctx.PassengerSet.FirstOrDefault();

        Console.WriteLine(p);

        foreach (var f in p.BookingSet)
        {
            Console.WriteLine("- hat Flug " + f.FlightNo + " gebucht!");
        }

    }
}

```

In this case, only two SQL commands were sent to the database management system. The fact that there were only two commands has nothing to do with the abandonment of the runtime proxies, but is due to the scenario that requires no more commands (Figure C-4).

```

Demo_LazyLoading_NoProxies
001:Debug #20100 Microsoft.EntityFrameworkCore.Database.Command.CommandExecuting:Executing DbCommand
[Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT TOP(1) [p].[PersonID], [p].[Birthday], [p].[CustomerSince], [p].[DetailID], [p].[EMail], [p].[
FrequentFlyer], [p].[GivenName], [p].[Status], [p].[Surname]
FROM [Passenger] AS [p]
#1: Julia Neumann
002:Debug #20100 Microsoft.EntityFrameworkCore.Database.Command.CommandExecuting:Executing DbCommand
[Parameters=[@__get_Item_0=?], CommandType='Text', CommandTimeout='30']
SELECT [e].[FlightNo], [e].[PassengerID]
FROM [Booking] AS [e]
WHERE [e].[PassengerID] = @__get_Item_0
- hat Flug 115 gebucht!
- hat Flug 122 gebucht!

```

Figure C-4. Output of Listing C-5

## Non-entity classes as a result of SQL queries

Starting with the Entity Framework Core 2.1, it is also possible to use `SqlQuery()` to map to classes that are not entity type. However, unlike the classic Entity Framework, in Entity Framework Core Version 2.1, you cannot just specify a remaining class in `FromSql()`.

Listing C-6 shows the class `FlightDTO`.

Listing C-6. A Data Transfer Object (DTO) for flights

```

using System;

namespace BO
{
    public class FlightDTO
    {
        public int FlightNo { get; set; }
        public string Departure { get; set; }
        public string Destination { get; set; }
        public DateTime Date { get; set; }

        public override string ToString()
        {
            return $"Flight {this.FlightNo}: {this.Departure}->{this.Destination} at {this.Date}";
        }
    }
}

```

This attempt

```
var flightSet = ctx.Set<FlightDTO>().FromSql("Select FlightNo, Departure,
Destination, FlightDate as Date from Flight");
```

leads to a runtime error

System.InvalidOperationException: 'Cannot create a dbSet for' FlightDTO 'because this type is not included in the model for the context.'

And this attempt

```
var flightSet = ctx.Query<FlightDTO>().FromSql("Select FlightNo, Departure,
Destination, FlightDate as Date from Flight");
```

leads to a runtime error

System.InvalidOperationException: 'Cannot create a dbSet for' FlightDTO 'because this type is not included in the model for the context.'

The use of `Query <T> ().FromSql ()` is correct, but unlike the classic Entity Framework, the developer must first log the type in the context class as `DbQuery <T>` (Listing C-7).

#### *Listing C-7.: Registering a query view in the context class*

```
public class WWWingsContext : DbContext
{

    #region Entities for tables
    public DbSet<Airline> AirlineSet { get; set; }
    public DbSet<Flight> FlightSet { get; set; }
    public DbSet<Pilot> PilotSet { get; set; }
    public DbSet<Passenger> PassengerSet { get; set; }
    public DbSet<Booking> BookingSet { get; set; }
    public DbSet<AircraftType> AircraftTypeSet { get; set; }
    #endregion

    #region Pseudo-entities for grouping results
    public DbSet<DepartureGrouping> DepartureGroupSet { get; set; } // for grouping
    result
    #endregion

    #region Pseudo-entities for views
    //public DbSet<DepartureStatistics> DepartureStatisticsSet { get; set; } // for
    view
    #endregion
}
```

```
#region Query Views

public DbQuery<BO.FlightDTO> FlightDTO { get; set; } // view

#endregion

...

}
```

**NOTE:** For types from Query Views, there is no change tracking with the Change Tracker and no way to persist changes.

By registering the DTO class with the context class, you can shorten the query instead of using `Query<T>()`:

```
var flightSet1 = ctx.FlightDTO.FromSql("Select FlightNo, Departure, Destination,
FlightDate as Date from Flight");
```

## Database Views

The mapping of database views is partially supported in Entity Framework Core only beginning with Entity Framework Core 2.1 Preview 1. Prior to that, there was only one workaround, treating views in program code like tables. However, this is a bit tricky, as Table C-1 shows.

*Table C-1. Mapping of database views.*

	<b>Entity Framework Core 1.0, 1.1, 2.0</b>	<b>Framework Core 2.1 Preview 1</b>
<b>Views during Reverse Engineering</b>	Unconsidered, can be used retrospectively with workaround.	Not taken into consideration, can be subsequently added manually as <code>DbQuery&lt;T&gt;</code> in the context class.
<b>Views in Forward Engineering</b>	Can be used with workaround if they were created manually.	Can be added as <code>dbQuery&lt;T&gt;</code> in the context class. The view must be created manually in the database.

**NOTE:** In Entity Framework Core 2.1 Preview 2, there is no change tracking for the data from views, nor is it possible to persist changes (unless you send SQL to the database by yourself).

## Create database view

The database view must be created manually in the database (possibly with the help of a tool such as SQL Server Management Studio) via `CREATE VIEW`. The database view created by `CREATE VIEW` in Listing C-8 provides the number of flights and the last flight for the flight table of the World Wide Wings database.

#### *Listing C-8. Creating a database view via SQL command*

```
USE WWWingsV2_EN
GO
CREATE VIEW dbo.[V_DepartureStatistics]
AS
SELECT departure, COUNT(FlightNo) AS FlightCount
FROM dbo.Flight
GROUP BY departure
GO
```

### Create the entity class for the database view

It is always necessary to create an entity class manually for the database view whose attributes correspond to the columns of the database view that are to be mapped (Class Departure Statistics include the data from the view "V\_ DepartureStatistics"). The [Table ] annotation specifies the database view name in the database because the entity class differs by name. Listing C-9 deliberately ignores the Last Flight column from the V\_PilotStatistics view.

For Entity Framework Core 1.0, 1.1, and 2.0, it is important that the entity class needs a primary key that can be specified using [Key] or the Fluent API HasKey () method.

#### *Listing C-9. An entity class with two properties for the two columns of the database view to be mapped*

```
[Table("V_DepartureStatistics")]
public class DepartureStatistics
{
    [Key] // must have a PK only in EFC 1.0, 1.1 and 2.0
    public string Departure { get; set; }
    public int FlightCount { get; set; }
}
```

An (artificial) key assignment is no longer necessary in Entity Framework Core version 2.1 or higher (Listing C-10).

#### *Listing C-10. Query View class with two properties for the two columns of the database view to be mapped*

```
[Table("V_DepartureStatistics")]
public class DepartureStatisticsView
{

```

```

public string Departure { get; set; }

public int FlightCount { get; set; }

}

```

## Including the entity class in the context class

Including the context class in the entity class differs in the Entity Framework Core Versions 1.0, 1.1 and 2.0 from the procedure in version 2.1.

### Entity Framework Core 1.0, 1.1 and 2.0

The entity class for the database view is now included as an entity class for a table in the context class via DbSet<T> (Listing C-11).

*Listing C-11. Including the entity class for the database view in the context class*

```

public class WWWingsContext: DbContext
{
    #region Entities for tables
    public DbSet<Airline> AirlineSet { get; set; }
    public DbSet<Flight> FlightSet { get; set; }
    public DbSet<Pilot> PilotSet { get; set; }
    public DbSet<Passenger> PassengerSet { get; set; }
    public DbSet<Booking> BookingSet { get; set; }
    public DbSet<AircraftType> AircraftTypeSet { get; set; }
    #endregion

    #region Pseudo-entities for views
    public DbSet<DepartureStatistics> DepartureStatisticsSet { get; set; } // for
view
    #endregion

    ...
}

```

### Entity Framework Core 2.1

The entity class for the database view is included in the context class as a query view via DbQuery<T> (Listing C-12).

*Listing C-12. Including the entity class for the database view in the context class*

```

public class WWWingsContext: DbContext
{
    #region Entities for tables

```

```

public DbSet<Airline> AirlineSet { get; set; }
public DbSet<Flight> FlightSet { get; set; }
public DbSet<Pilot> PilotSet { get; set; }
public DbSet<Passenger> PassengerSet { get; set; }
public DbSet<Booking> BookingSet { get; set; }
public DbSet<AircraftType> AircraftTypeSet { get; set; }
#endregion

#region Query Views
public DbQuery<DepartureStatisticsView> DepartureStatisticsView { get; set; }
// View
#endregion
...
}

```

## Use of the database view

The use of the view is now identical for Entity Framework Core 1.x/2.0 and from version 2.1 (if you use the same name). As an illustration, however, different names have been used.

The `dbSet<T>` or the `dbQuery<T>` for the database view (Listing C-13) can now be used by the software developer, such as the entity class for a table in LINQ queries, or `FromSql()` for direct SQL queries.

**NOTE:** `DbQuery<T>` always returns context-detached data that is not monitored by the Change Tracker and thus cannot be changed.

In the case of the trick with `dbSet<T>`, if the database view is writable, the contents of the Entity Framework Core API could also be changed via `SaveChanges()`, respectively, append and delete records.

### *Listing C-13. Use of the entity class for the database view*

```

using DA;

using ITVisions;

using System;

using System.Linq;

namespace EFC_Console
{
    public class DatabaseViews

```

```

{

    /// <summary>
    /// Read Data from View
    /// EFC 1.x/2.0
    /// </summary>
    [EFCBook]
    public static void DatabaseViewWithPseudoEntity()
    {
        CUI.MainHeadline(nameof(DatabaseViewWithPseudoEntity));

        using (var ctx = new WWWingsContext())
        {
            var query = ctx.DepartureStatisticsSet.Where(x => x.FlightCount > 0);
            var liste = query.ToList();
            foreach (var stat in liste)
            {
                Console.WriteLine($"{stat.FlightCount:000} Flights departing from
{stat.Departure}.");
            }
        }
    }

    /// Read Data from View
    /// EFC >= 2.1>
    [EFCBook]
    public static void DatabaseView()
    {
        CUI.MainHeadline(nameof(DatabaseView));

        using (var ctx = new WWWingsContext())
        {
            // Composition SQL on VIEW :->
            var query = ctx.DepartureStatisticsView.Where(x => x.FlightCount >
0).OrderBy(x => x.FlightCount);
            var liste = query.ToList();
            foreach (var stat in liste)
            {
                Console.WriteLine($"{stat.FlightCount:000} Flights departing from
{stat.Departure}.");
            }
        }
    }
}

```



```

    }
}
}
}

```

## Challenge: Migrations

The use of `DbSet<T>` seems to have the advantage of being able to change the view data, but it also has a major disadvantage: If you create a schema migration in the context class after creating the database view, you will notice that Entity Framework Core is now unwantedly creating a table for the database view in the database (Listing C-14).

*Listing C-14. Entity Framework Core creates a `CreateTable()` for the database view in the schema migration class, which is not desirable.*

```

using Microsoft.EntityFrameworkCore.Migrations;
using System;
using System.Collections.Generic;

namespace DA.Migrations
{
    public partial class v8 : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {

            migrationBuilder.CreateTable(
                name: "V_DepartureStatistics",
                columns: table => new
                {
                    Departure = table.Column<string>(nullable: false),
                    FlightCount = table.Column<int>(nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_V_DepartureStatistics", x =>
x.Departure);
                });
        }

        protected override void Down(MigrationBuilder migrationBuilder)
        {

```

```

        migrationBuilder.DropTable(
            name: "V_DepartureStatistics");
    }
}
}

```

This is correct from the point of view of Entity Framework Core, as we told the ORM that "V\_DepartureStatistics" would be a table.

Unfortunately, this schema migration could not be executed because there can only be one object named "V\_DepartureStatistics" in the database.

There are two possible solutions to this situation:

- You manually delete the CreateTable() in the Up() method and the corresponding DropTable() in Down() from the migration class.
- Entity Framework Core is tricked, so that the ORM ignores the entity class DepartureStatistics when creating the migration step at development time, but not at runtime.

**NOTE:** The trick is realized in Listing C-15. Entity Framework Core instantiates the context class as part of creating or deleting a schema migration and calls OnModelCreating(). However, this does not happen at development time via the actual starting point of the application (then the application would actually start), but by hosting the DLL with the context class in the command line tool ef.exe. In OnModelCreating(), you therefore check whether the current process has the name "ef ". If so, then we are not in the running time of the application, but in the development environment and want to ignore the database view with Ignore(). At runtime of the application, however, the Ignore() will not be executed, and thus using the database view through the entity class is possible.

*Listing C-15. Entity Framework Core should only ignore the entity class for the database view at development time.*

```

protected override void OnModelCreating (ModelBuilder modelBuilder)
{
    // Trick: hide the database view from the EF Migration Tool, so it does not
    // want to create a new table for it
    if (System.Diagnostics.Process.GetCurrentProcess().ProcessName.ToLower() ==
        "ef")
    {
        modelBuilder.Ignore <DepartureStatistics>();
    }
    ...
}

```

```
}
```

**NOTE:** There is an alternative trick. If the query of the process name is too uncertain, because Microsoft could change this name, you can instead use a switch in the context class in the form of a static attribute (e.g., `bool IsRuntime { get; set; } = false`). By default, this `IsRuntime` is false and ignores the entity class for the database view. At runtime, however, the application sets `IsRuntime` to true before the first instantiation of the context class.

## Value Conversions (Value Converter)

In the classic Entity Framework and Entity Framework Core 1.0, 1.1, 2.0, there was one compatible .NET data type for each database column type. Although it was possible to force a different database column type with data annotation [`Column (TypeName = "typename")`] or in the Fluent API, this only worked insofar as the types were compatible from the point of view of the database management system. Conversion/conversion of values during loading (materializing) or saving new and changed data records was not planned.

In the Entity Framework Core 2.1 Preview 1, Microsoft introduced Value Converter to realize such a value conversion. The conversion is defined in the Fluent API with `HasConversion()`. There are predefined converters (Figure C-5) and you can create your own converter.

- `BoolToZeroOneConverter` - Bool to zero and one
- `BoolToStringConverter` - Bool to strings such as "Y" and "N"
- `BoolToTwoValuesConverter` - Bool to any two values
- `BytesToStringConverter` - Byte array to Base64-encoded string
- `CastingConverter` - Conversions that require only a Csharp cast
- `CharToStringConverter` - Char to single character string
- `DateTimeOffsetToBinaryConverter` - DateTimeOffset to binary-encoded 64-bit value
- `DateTimeOffsetToBytesConverter` - DateTimeOffset to byte array
- `DateTimeOffsetToStringConverter` - DateTimeOffset to string
- `DateTimeToBinaryConverter` - DateTime to 64-bit value including DateTimeKind
- `DateTimeToStringConverter` - DateTime to string
- `DateTimeToTicksConverter` - DateTime to ticks
- `EnumToNumberConverter` - Enum to underlying number
- `EnumToStringConverter` - Enum to string
- `GuidToBytesConverter` - Guid to byte array
- `GuidToStringConverter` - Guid to string
- `NumberToBytesConverter` - Any numerical value to byte array
- `NumberToStringConverter` - Any numerical value to string
- `StringToBytesConverter` - String to UTF8 bytes
- `TimeSpanToStringConverter` - TimeSpan to string
- `TimeSpanToTicksConverter` - TimeSpan to ticks

Figure C-5. List of predefined converters (<https://docs.microsoft.com/en-us/ef/core/modeling/value-conversions>)

## Constraints

The conversion is never called for NULL values. The conversion only refers to a property/database table column; it is not possible to convert several properties/database table columns together. In the Preview 1 version, some conversions still do not work correctly.

### Example 1: Conversion between String and Boolean

The entity class `Passenger` should receive an additional Property `FrequentFlyer`, which is represented in the program code as string ("Yes" and "No"), but in the database more efficiently as Boolean or Bit should be stored.

```
public partial class Passenger : Person
{
```

```
...
    public bool FrequentFlyer { get; set; }
}
```

In `OnModelCreating()` in the context class, define a converter for this property. Since there is no Microsoft-predefined converter for this case, the developer must define a converter. There are two syntactic options:

- Instantiating the `ValueConverter` class and setting two lambda expressions for both conversion directions in the constructor (`ConvertToStoreExpression`: the object-to-record conversion when saving is the first constructor parameter; `ConvertFromStoreExpression`: the second constructor parameter describes the conversion from the record to the object during materialization)
- Use of the two lambda expressions directly in the method `HasConversion()`

*Listing C-16. Converter between String and Boolean*

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // FrequentFlyer String<->Bool Converter: Syntax 1: Explicit Converter
instance
    var converterForFrequentFlyer = new ValueConverter<string, bool>(
        v => (v == "Yes" ? true : false),
        v => (v ? "Yes" : "No"));
    modelBuilder
        .Entity<Passenger>().Property(x => x.FrequentFlyer).
        HasConversion(converterForFrequentFlyer);

    // FrequentFlyer String<->Bool Converter: Alternative Syntax 2: Inline
Converter
    modelBuilder
        .Entity<Passenger>().Property(x => x.FrequentFlyer).
        HasConversion(v => (v == "Yes" ? true : false), v => (v ? "Yes" : "No"));
}
```

In the database schema migrations, you can see that the Entity Framework Core now actually has the column `FrequentFlyer` as Boolean value (i.e., SQL Server data type "bit") (Listing C-17 and Figure C-6).

*Listing C-17. Schema migration for the new FrequentFlyer column*

```
using System;
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore.Migrations;
```

```

namespace DA.Migrations
{
    public partial class v3 : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.AddColumn<bool>(
                name: "FrequentFlyer",
                table: "Passenger",
                nullable: true);

        }

        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropColumn(
                name: "FrequentFlyer",
                table: "Passenger");

        }
    }
}

```



	Column Name	Data Type	Allow Nulls
	PersonID	int	<input type="checkbox"/>
	Surname	nvarchar(MAX)	<input checked="" type="checkbox"/>
	GivenName	nvarchar(MAX)	<input checked="" type="checkbox"/>
	Birthday	datetime2(7)	<input checked="" type="checkbox"/>
	EMail	nvarchar(MAX)	<input checked="" type="checkbox"/>
	DetailID	int	<input checked="" type="checkbox"/>
	CustomerSince	datetime2(7)	<input checked="" type="checkbox"/>
	FrequentFlyer	bit	<input checked="" type="checkbox"/>
	Status	nvarchar(1)	<input type="checkbox"/>

Figure C-6. The database creates a bit column for FrequentFlyer.

Listing C-18 shows a test case for this feature.

*Listing C-18. Using the new FrequentFlyer column*

```
/// <summary>
    /// Test for FrequentFlyer Property
    /// </summary>
    [EFCBook("5.0", "2.1")]
    public static void ConvertStringToBoolean()
    {
        CUI.MainHeadline(nameof(ConvertStringToBoolean));
        using (WWWingsContext ctx = new WWWingsContext())
        {
            ctx.Log();

            // Add new Passenger
            var p = new Passenger();
            p.GivenName = "Max";
            p.Surname = "Müller";
            p.Birthday = new DateTime(2070, 2, 3);
            p.Status = 'A';
            p.FrequentFlyer = "Yes";
            ctx.PassengerSet.Add(p);
            var count = ctx.SaveChanges();

            Console.WriteLine("Saved Changes: " + count);
            Console.WriteLine("Added new Passenger #" + p.PersonID);

            // Get all Frequent Travellers
            Console.WriteLine("All Frequent Travellers:");
            var ft = ctx.PassengerSet.Where(x => x.FrequentFlyer == "Yes").ToList();
            foreach (var pas in ft)
            {
                Console.WriteLine(pas);
            }

            // Get all Frequent Travellers
            Console.WriteLine("All Frequent Travellers:");
            var ft2 = ctx.PassengerSet.Where(x =>
x.FrequentFlyer.StartsWith("Y")).ToList();
            foreach (var pas in ft2)
```

```

{
    Console.WriteLine(pas);
}

// Get raw data from Database as DataReader
var r = ctx.Database.ExecuteSqlQuery("Select p.PersonID, p.Surname,
p.Birthday, p.Status from Passenger as p where p.personID= " + p.PersonID);
DbDataReader dr = r.DbDataReader;
while (dr.Read())
{
    Console.WriteLine("{0}\t{1}\t{2}\t{3} \n", dr[0], dr[1], dr[2], dr[3]);
}
dr.Dispose();
}

```

In Listing C-18, a passenger is first created as a frequent flyer and then all frequent flyers are read out. Entity Framework Core sends to the database:

```

SELECT [x].[PersonID], [x].[Birthday], [x].[CustomerSince], [x].[DetailID],
[x].[EMail], [x].[FrequentFlyer], [x].[GivenName], [x].[Status], [x].[Surname]
FROM [Passenger] AS [x]
WHERE [x].[FrequentFlyer] = 1

```

However, converters also present challenges. When formulating in LINQ:

```
var ft = ctx.PassengerSet.Where(x => x.FrequentFlyer.StartsWith("Y")).ToList();
```

sends Entity Framework Core to the database:

```

SELECT [x].[PersonID], [x].[Birthday], [x].[CustomerSince], [x].[DetailID],
[x].[EMail], [x].[FrequentFlyer], [x].[GivenName], [x].[Status], [x].[Surname]
FROM [Passenger] AS [x]
WHERE [x].[FrequentFlyer] LIKE 0 + 0 AND (LEFT([x].[FrequentFlyer], LEN(N'Y')) =
0)

```

This does not provide those who have a 1 in the column FrequentFlyer, but those who have a 0!

## Example 2: Conversion between enumeration type and string

**NOTE:** This example shows a case that, according to the documentation, should work, but actually does not work in several places in the Preview 1 version.

The enumeration type PilotLicenseType in the pilot class should not be persisted as a numerical value, but now as a string.



The class Pilot cannot be changed: It still has a property PilotLicenseType for the enumeration type PilotLicenseType (Listing C-19).

*Listing C-19. Pilot class with PilotLicenseType*

```
using System;

using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace BO
{

    public enum PilotLicenseType
    {
        // https://en.wikipedia.org/wiki/Pilot_licensing_and_certification
        Student, Sport, Recreational, Private, Commercial, FlightInstructor, ATP
    }

    [Serializable]
    public partial class Pilot : Employee
    {
        // PK is inherited from Employee

        #region Primitive Properties
        public virtual DateTime LicenseDate { get; set; }
        public virtual Nullable<int> FlightHours { get; set; }

        public virtual PilotLicenseType PilotLicenseType
        {
            get;
            set;
        }
        ...
    }
}
```

In `OnModelCreating()` in the context class a converter has to be set for this property. There are three syntactic options (Listing C-20):

- Use one of the predefined converter class `EnumToStringConverter`
- Instantiate the `ValueConverter` class and set two lambda expressions for both conversion directions in the constructor (`convertToStoreExpression`: the object-to-record conversion when saving is the first constructor parameter; `convertFromStoreExpression`: the second constructor parameter describes the conversion from the record to the object during materialization)
- Use two lambda expressions directly in the method `HasConversion()`

*Listing C-20. Set converter for `PilotLicenseType`*

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // Syntax 1: Predefined Converter instance
    var converter = new EnumToStringConverter<PilotLicenseType>();
    modelBuilder
        .Entity<Pilot>()
        .Property(e => e.PilotLicenseType)
        .HasConversion(converter);

    // Alternative Syntax 2: Explicit Converter instance
    var valueconverter = new ValueConverter<PilotLicenseType, string>(
        v => v.ToString(),
        v => (PilotLicenseType)Enum.Parse(typeof(PilotLicenseType), v));

    modelBuilder
        .Entity<Pilot>().Property(x => x.PilotLicenseType).
        HasConversion(valueconverter);

    // Alternative Syntax 3: Inline Converter
    modelBuilder
        .Entity<Pilot>().Property(x => x.PilotLicenseType).
        HasConversion(x => x.ToString(),
            x => (PilotLicenseType)Enum.Parse(typeof(PilotLicenseType), x));
    ...
}
```

In a migration class created with `Add Migration`, the property presents itself as shown in Listing C-21 (the table name is called `Employee`, as `Pilot` of `Employee` and `Entity Framework Core` uses table-by-hierarchy mapping for inheritance).

#### *Listing C-21. migration class for PilotLicenseType*

```
using System;

using System.Collections.Generic;
using Microsoft.EntityFrameworkCore.Migrations;

namespace DA.Migrations
{
    public partial class v3 : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.AddColumn<string>(
                name: "PilotLicenseType",
                table: "Employee",
                nullable: true);
        }

        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropColumn(
                name: "PilotLicenseType",
                table: "Employee");
        }
    }
}
```

For update database response script migration, there will be an error: "Converter for model type 'PilotLicenseType' cannot be used for 'BO.Pilot.PilotLicenseType' because its type is 'int'."

The same thing happens with the EnumToStringConverter example of the Microsoft documentation. I have reported this as a bug to Microsoft: <https://github.com/aspnet/EntityFrameworkCore/issues/11338>.

The database currently can only be created at runtime with this converter by calling the method `Ctx.Database.EnsureCreated()`.

Note also the differences in the data type length: Syntax form 1 creates `nvarchar (512)`, because the maximum length for identifier names in C # is 511 characters. In syntax forms 2 and 3, the

term could theoretically give rise to longer names. Therefore, Entity Framework Core creates `nvarchar(MAX)` with caution.

For syntax form 2, you can control the length using the third constructor parameter of the `ValueConverter` class. Listing C-22 shows the limit of 20 characters using an instance of `ConverterMappingHints`.

*Listing C-22. A limit of 20 characters*

```
var cmh = new ConverterMappingHints(20);  
var valueconverter = new ValueConverter<PilotLicenseType, string>(  
    v => v.ToString(),  
    v => (PilotLicenseType)Enum.Parse(typeof(PilotLicenseType), v), cmh);  
modelBuilder  
    .Entity<Pilot>().Property(x => x.PilotLicenseType).  
    HasConversion(valueconverter);
```

This limits the string length of the column `PilotLicenseType` to `nvarchar(20)` as shown in Figure C-7.

	Column Name	Data Type	Allow Nulls
🔑	PersonID	int	<input type="checkbox"/>
	Surname	nvarchar(MAX)	<input checked="" type="checkbox"/>
	GivenName	nvarchar(MAX)	<input checked="" type="checkbox"/>
	Birthday	datetime2(7)	<input checked="" type="checkbox"/>
	EMail	nvarchar(MAX)	<input checked="" type="checkbox"/>
	DetailID	int	<input checked="" type="checkbox"/>
	Salary	real	<input type="checkbox"/>
	SupervisorPersonID	int	<input checked="" type="checkbox"/>
	PassportNumber	nvarchar(MAX)	<input checked="" type="checkbox"/>
	Discriminator	nvarchar(MAX)	<input type="checkbox"/>
	LicenseDate	datetime2(7)	<input checked="" type="checkbox"/>
	FlightHours	int	<input checked="" type="checkbox"/>
▶	PilotLicenseType	nvarchar(20)	<input checked="" type="checkbox"/>

*Figure C-7. ConverterMappingHints leads to length limitation*

Listing C-23 shows the use of the conversion when creating a pilot and when reading out all pilots with a certain type of ticket. Further reading of the data record with the `DataReader` at the end is used to check whether the character string has actually been stored in the database instead of the number of the enumerated numerical value.

*Listing C-23. Program code that uses converter*

```
public static void ConvertEnumToString()
{
    CUI.MainHeadline(nameof(ConvertEnumToString));
    using (WWWingsContext ctx = new WWWingsContext())
    {
        // Add new Pilot
        var p = new Pilot();
        p.GivenName = "Max";
        p.Surname = "Müller";
        p.Birthday = new DateTime(2070, 2, 3);
        p.PilotLicenseType = PilotLicenseType.FlightInstructor;
        ctx.PilotSet.Add(p);
        var count = ctx.SaveChanges();

        Console.WriteLine("Saved Changes: " + count);
        Console.WriteLine("Added new Pilot #" + p.PersonID);

        // Get raw data from Database as DataReader
        var r = ctx.Database.ExecuteSqlQuery("Select p.PersonID, p.Surname,
p.PilotLicenseType, p.Birthday from Employee as p where p.personID= " +
p.PersonID);
        DbDataReader dr = r.DbDataReader;
        while (dr.Read())
        {
            Console.WriteLine("{0}\t{1}\t{2}\t{3} \n", dr[0], dr[1], dr[2], dr[3]);
        }
        dr.Dispose();
    }
}
```

Indeed, the data access with LINQ does not work. Entity Framework Core sends the number value instead of the string to the database:

```
SELECT [x].[PersonID], [x].[Birthday], [x].[DetailID], [x].[Discriminator],
[x].[EMail], [x].[GivenName], [x].[PassportNumber], [x].[Salary],
[x].[SupervisorPersonID], [x].[Surname], [x].[FlightHours], [x].[FlightSchool],
[x].[LicenseDate], [x].[PilotLicenseType]
```

```
FROM [Employee] AS [x]
WHERE ([x].[Discriminator] = N'Pilot') AND ([x].[PilotLicenseType] = 5)
```

The database then exits with the following runtime error: "System.Data.SqlClient.SqlException: 'Conversion failed when converting the nvarchar value 'FlightInstructor' to data type int."

## Transaction across multiple context instances with TransactionScope

This section shows the use of `System.Transactions.TransactionScope` for a transaction over two executions of `SaveChanges()` in two different context instances. This was possible in the classic Entity Framework and is again possible in Entity Framework Core from version 2.1.

Using the `TransactionScope` class provides a declarative way to execute specific blocks of code in a transaction. The developer uses them, for example, by instantiating them in a block and performing the transactional tasks within the block. In this case, all commands sent to a database via ADO.NET in the code block become part of the transaction, including commands sent via Entity Framework or Entity Framework Core. Other transactional resources such as the Transactional File System (TxF) and the Transactional Registry (TxR), both of which have been available since Windows Vista or Windows Server 2008, may be part of the transaction (for TxF and TxR, special Win32 API calls are required, which are the .NET standard classes in the naming `system.IO` or `3 Microsoft 20.131 00:33:3Win32.Registry` not realize).

To benefit from the class `TransactionScope`, the developer binds the Assembly `System.Transactions.dll`. Ideally, this class is used within a using statement. Thus, the beginning and the end of the transaction represented are clearly outlined. Access to transactional resources within a `TransactionScope` area is done as part of a transaction. To commit the transaction, the developer calls the `Complete()` method at the end of the scope. To roll back the transaction, the developer omits this and leaves the block differently (e.g., by return and a runtime error).

If the developer accesses more than one transactional resource within a `TransactionScope` scope, for example, two different databases will attempt to start a distributed transaction. In addition to the fact that this is accompanied by a not insignificant overhead, all involved resources must support distributed transactions. The Distributed Transaction Coordinator (DTC) delivered with Windows in the form of a system service must be started on the affected computers.

On the other hand, the developer accesses one transactional resource via two different connections (e.g., if there are two database connections to the same database), and the transaction behavior depends on the database driver used. In this case, if the database driver supports the Lightweight Transaction Manager (such as the Microsoft SQL Server driver), a local transaction is used and the DTC services are not needed. This is the case in Listing C-24.

#### *Listing C-24. Transaction with TransactionScope*

```
public static void TransactionScopeDemo()
{
    CUI.Headline(nameof(TransactionScopeDemo));

    using (var t = new TransactionScope())
    {
        using (var ctx1 = new WWWingsContext())
        {
            int flightNo = ctx1.FlightSet.OrderBy(x =>
x.FlightNo).FirstOrDefault().FlightNo;
            var f = ctx1.FlightSet.Where(x => x.FlightNo == flightNo).SingleOrDefault();

            Console.WriteLine("Before: " + f.ToString());
            f.FreeSeats--;
            f.Memo = "Last changed at " + DateTime.Now.ToString();

            Console.WriteLine("After: " + f.ToString());

            var count1 = ctx1.SaveChanges();
            Console.WriteLine("Number of saved changes: " + count1);
        }

        using (var ctx2 = new WWWingsContext())
        {
            var f = ctx2.FlightSet.OrderBy(x =>
x.FlightNo).Skip(1).Take(1).SingleOrDefault();

            Console.WriteLine("Before: " + f.ToString());
            f.FreeSeats--;
            f.Memo = "Last changed at " + DateTime.Now.ToString();

            Console.WriteLine("After: " + f.ToString());

            var count1 = ctx2.SaveChanges();
            Console.WriteLine("Number of saved changes: " + count1);
        }

        // Commit Transaction!
        t.Complete();
    }
}
```

```
CUI.PrintSuccess("Completed!");  
  
}  
  
}
```

The `TransactionScope` class also allows transactions to be nested. If another instance of `TransactionScope` is created within a block that owns a `TransactionScope` instance, the `TransactionScopeOption` parameter must specify the ratio of the transactions. Possible values are `Required`, `RequiresNew` and `Suppress`.