Entity Framework Core 7 (EF7)

Entity Framework Core (EF Core) 7 is available on NuGet today!

EF Core 7 is the successor to EF Core 6, and can be referred to as EF7 for brevity. EF Core 7 contains many features that help in porting "classic" EF6 applications to use EF7. As such, we encourage people to upgrade existing classic EF applications to use EF7 where possible. See Porting from EF6 to EF Core for more information.

TIP EF7 is released under the standard support policy, but works on both the long-term-support .NET 6 and the standard-support .NET 7. EF7 does not run on .NET Framework.

The following sections give an overview of the major enhancements in EF7. In total, EF7 ships with 167 enhancements and new features, both large and small, as well as 171 bug fixes.

TIP Full details of all new EF7 features can be found in the What's New in EF7 documentation. All the code is available in runnable samples on GitHub.

JSON columns

Most relational databases support columns that contain JSON documents, and the JSON in these columns can be drilled into with queries. This allows, for example, filtering and sorting by properties inside the documents, as well as projection of properties out of the documents into results. JSON columns allow relational databases to take on some of the characteristics of document databases, creating a useful hybrid between the two; they can also be used to eliminate joins from queries, improving performance.

EF7 contains provider-agnostic support for JSON columns, with an implementation for SQL Server. This support allows mapping of aggregates built from .NET types to JSON documents. Normal LINQ queries can be used on the aggregates, and these are translated to the appropriate query constructs needed to drill into the JSON. EF7 also supports saving changes to the JSON documents.

Using LINQ to query JSON

Consider the following LINQ query:

```
var postsWithViews = await context.Posts.Where(post => post.Metadata!.Views > 3000)
    .AsNoTracking()
    .Select(
        post => new
        {
            post.Author!.Name,
            post.Metadata!.Views,
            Searches = post.Metadata.TopSearches,
            Commits = post.Metadata.Updates
        })
    .ToListAsync();
```

EF7 translates this query into the following SQL:

```
SELECT [a].[Name],
      CAST(JSON_VALUE([p].[Metadata],'$.Views') AS int),
      JSON_QUERY([p].[Metadata],'$.TopSearches'),
      [p].[Id],
      JSON_QUERY([p].[Metadata],'$.Updates')
FROM [Posts] AS [p]
LEFT JOIN [Authors] AS [a] ON [p].[AuthorId] = [a].[Id]
WHERE CAST(JSON_VALUE([p].[Metadata],'$.Views') AS int) > 3000
```

Notice that the JSON_VALUE and JSON_QUERY are used to query into parts of the JSON document.

Updating JSON with SaveChanges

EF7's change tracking finds the smallest single part of a JSON document that needs to be updated and sends SQL commands to efficiently update the column appropriately. For example, consider code that modifies a single property embedded inside a JSON document:

```
var arthur = await context.Authors.SingleAsync(author => author.Name.StartsWith("Arthur"));

arthur.Contact.Address.Country = "United Kingdom";

await context.SaveChangesAsync();
```

EF7 generates a SQL parameter for only the changed value:

```
@p0='["United Kingdom"]' (Nullable = false) (Size = 18)
```

And then uses this parameter with the JSON_MODIFY command:

```
UPDATE [Authors] SET [Contact] = JSON_MODIFY([Contact], 'strict $.Address.Country',
JSON_VALUE(@p0, '$[0]'))
OUTPUT 1
WHERE [Id] = @p1;
```

More about JSON columns

To find out more about mapping JSON columns in EF7, see:

- What's New: JSON Columns
- .NET Data Community Standup Video: JSON Columns
- Sample code: JSON Columns

Bulk updates and deletes

EF Core tracks changes to entities and then sends updates to the database when SaveChangesAsync is called. Changes are only sent for properties and relationships that have actually changed. Also, the tracked entities remain in sync with the changes sent to the database. This mechanism is an efficient and convenient way to send general-purpose inserts, updates, and deletes to the database. These changes are also batched to reduce the number of database roundtrips.

However, it is sometimes useful to execute update or delete commands on the database without loading the entities or involving the change tracker. EF7 enables this with the new ExecuteUpdateAsync and ExecuteDeleteAsync methods. These methods are applied to a LINQ query and update or delete entities in the database immediately based on the results of that query. Many entities can be updated with a single command and the entities are not loaded into memory.

Bulk delete

Consider the following LINQ query terminated with a call to ExecuteDeleteAsync:

```
var priorToDateTime = new DateTime(priorToYear, 1, 1);
```

```
await context.Tags.Where(t => t.Posts.All(e => e.PublishedOn <
priorToDateTime)).ExecuteDeleteAsync();
```

This generates SQL to immediately delete from the database all tags for posts published before the given year:

```
DELETE FROM [t]
FROM [Tags] AS [t]
WHERE NOT EXISTS (
    SELECT 1
    FROM [PostTag] AS [p]
    INNER JOIN [Posts] AS [p0] ON [p].[PostsId] = [p0].[Id]
    WHERE [t].[Id] = [p].[TagsId] AND [p0].[PublishedOn] < @__priorToDateTime_1)
```

Bulk update

Using ExecuteUpdateAsync is very similar to using ExecuteDeleteAsync, except that it requires additional arguments to specify the changes to make to each row. For example, consider the following LINQ query terminated with a call to ExecuteUpdateAsync:

```
var priorToDateTime = new DateTime(priorToYear, 1, 1);


await context.Tags
    .Where(t => t.Posts.All(e => e.PublishedOn < priorToDateTime))
    .ExecuteUpdateAsync(s => s.SetProperty(t => t.Text, t => t.Text + " (old)"));
```

This generates SQL to immediately update the “Text” column of all tags for posts published before the given year:

```
UPDATE [t]
    SET [t].[Text] = [t].[Text] + N' (old)'
FROM [Tags] AS [t]
WHERE NOT EXISTS (
    SELECT 1
    FROM [PostTag] AS [p]
    INNER JOIN [Posts] AS [p0] ON [p].[PostsId] = [p0].[Id]
    WHERE [t].[Id] = [p].[TagsId] AND [p0].[PublishedOn] < @__priorToDateTime_1)
```

Update/delete for single rows

While ExecuteUpdateAsync and ExecuteDeleteAsync are commonly used to update or delete many rows at the same time (i.e. “bulk” changes), they can

also be useful for efficient single row changes. For example, consider the common pattern to delete an entity in an ASP.NET Core application:

```
public async Task<ActionResult> DeletePost(int id)
{
    var post = await _context.Posts.FirstOrDefaultAsync(p => p.Id == id);

    if (post == null)
    {
        return NotFound();
    }

    _context.Posts.Remove(post);
    await _context.SaveChangesAsync();

    return Ok();
}
```

This can be changed when using EF7 to:

```
public async Task<ActionResult> DeletePost(int id)
    => await _context.Posts.Where(p => p.Id == id).ExecuteDeleteAsync() == 0
        ? NotFound()
        : Ok();
```

This is both less code and is significantly faster since it performs only a single database round-trip.

When to use bulk updates

ExecuteUpdateAsync and ExecuteDeleteAsync are a great choice for simple, well-specified updates and deletes. However, keep in mind that:

- The specific changes to make must be specified explicitly; they are not automatically detected by EF Core.
- Any tracked entities are not kept in sync.
- Multiple commands may be needed, and these need to be in the correct order so as not to violate database constraints. For example, dependents must be deleted before a principal can be deleted.
- Multiple calls to ExecuteUpdateAsync and ExecuteDeleteAsync not be automatically wrapped in a transaction.

All of this means that ExecuteUpdateAsync and ExecuteDeleteAsync complement, rather than replace, the existing SaveChanges mechanism.

More about bulk updates

To find out more about ExecuteUpdateAsync and ExecuteDeleteAsync in EF7, see:

- What's New: ExecuteUpdate and ExecuteDelete (Bulk updates)
- .NET Data Community Standup Video: Bulk updates
- Sample code: ExecuteUpdateAsync
- Sample code: ExecuteDeleteAsync

Faster SaveChanges

In EF7, the performance of SaveChanges and SaveChangesAsync has been significantly improved. In some scenarios, saving changes is now four times faster than with EF Core 6. These improvements come from performing fewer roundtrips to the database and generation of more efficient SQL.

Elimination of unneeded transactions

All modern relational databases guarantee transactionality for (most) single SQL statements. That is, the statement will never be only partially completed, even if an error occurs. EF7 avoids starting an explicit transaction in these cases.

For example, consider the following call to SaveChangesAsync which inserts a single entity:

```
await context.AddAsync(new Blog { Name = "MyBlog" });
await context.SaveChangesAsync();
```

In EF Core 6, the INSERT command is wrapped by commands to begin and then commit a transaction:

```
dbug: 9/29/2022 11:43:09.196 RelationalEventId.TransactionStarted[20200]
(Microsoft.EntityFrameworkCore.Database.Transaction)
      Began transaction with isolation level 'ReadCommitted'.
info: 9/29/2022 11:43:09.265 RelationalEventId.CommandExecuted[20101]
(Microsoft.EntityFrameworkCore.Database.Command)
```

```
      Executed DbCommand (27ms) [Parameters=[@p0='MyBlog' (Nullable = false) (Size = 4000)],
CommandType='Text', CommandTimeout='30']
      SET NOCOUNT ON;
      INSERT INTO [Blogs] ([Name])
      VALUES (@p0);
      SELECT [Id]
      FROM [Blogs]
      WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
dbug: 9/29/2022 11:43:09.297 RelationalEventId.TransactionCommitted[20202]
(Microsoft.EntityFrameworkCore.Database.Transaction)
      Committed transaction.
```

EF7 detects that the transaction is not needed here and so removes these calls:

```
info: 9/29/2022 11:42:34.776 RelationalEventId.CommandExecuted[20101]
(Microsoft.EntityFrameworkCore.Database.Command)
      Executed DbCommand (25ms) [Parameters=[@p0='MyBlog' (Nullable = false) (Size = 4000)],
CommandType='Text', CommandTimeout='30']
      SET IMPLICIT_TRANSACTIONS OFF;
      SET NOCOUNT ON;
      INSERT INTO [Blogs] ([Name])
      OUTPUT INSERTED.[Id]
      VALUES (@p0);
```

Inserting multiple rows

In EF Core 6, the default approach for inserting multiple rows was driven by limitations in SQL Server support for tables with triggers. This mans that EF Core 6 could not use a simple OUTPUT clause. Instead, when inserting multiple entities, EF Core 6 generated some fairly convoluted SQL involving a temporary table. For example, consider this call to SaveChangesAsync:

```
for (var i = 0; i < 4; i++)
{
    await context.AddAsync(new Blog { Name = "Foo" + i });
}

await context.SaveChangesAsync();
```

The SQL generated by EF Core 6 is the following:

```
dbug: 9/30/2022 17:19:51.919 RelationalEventId.TransactionStarted[20200]
(Microsoft.EntityFrameworkCore.Database.Transaction)
      Began transaction with isolation level 'ReadCommitted'.
info: 9/30/2022 17:19:51.993 RelationalEventId.CommandExecuted[20101]
(Microsoft.EntityFrameworkCore.Database.Command)
      Executed DbCommand (27ms) [Parameters=[@p0='Foo0' (Nullable = false) (Size = 4000),
@p1='Foo1' (Nullable = false) (Size = 4000), @p2='Foo2' (Nullable = false) (Size = 4000),
@p3='Foo3' (Nullable = false) (Size = 4000)], CommandType='Text', CommandTimeout='30']
      SET NOCOUNT ON;
      DECLARE @inserted0 TABLE ([Id] int, [_Position] [int]);
      MERGE [Blogs] USING (
      VALUES (@p0, 0),
      (@p1, 1),
      (@p2, 2),
      (@p3, 3)) AS i ([Name], _Position) ON 1=0
      WHEN NOT MATCHED THEN
      INSERT ([Name])
      VALUES (i.[Name])
      OUTPUT INSERTED.[Id], i._Position
      INTO @inserted0;

      SELECT [i].[Id] FROM @inserted0 i
      ORDER BY [i].[_Position];
dbug: 9/30/2022 17:19:52.023 RelationalEventId.TransactionCommitted[20202]
(Microsoft.EntityFrameworkCore.Database.Transaction)
      Committed transaction.
```

In contrast, EF7 generates a single, simpler command when targeting a table without triggers:

```
info: 9/30/2022 17:40:37.612 RelationalEventId.CommandExecuted[20101]
(Microsoft.EntityFrameworkCore.Database.Command)
      Executed DbCommand (4ms) [Parameters=[@p0='Foo0' (Nullable = false) (Size = 4000),
@p1='Foo1' (Nullable = false) (Size = 4000), @p2='Foo2' (Nullable = false) (Size = 4000),
@p3='Foo3' (Nullable = false) (Size = 4000)], CommandType='Text', CommandTimeout='30']
      SET IMPLICIT_TRANSACTIONS OFF;
      SET NOCOUNT ON;
```

```
MERGE [Blogs] USING (
VALUES (@p0, 0),
(@p1, 1),
(@p2, 2),
(@p3, 3)) AS i ([Name], _Position) ON 1=0
WHEN NOT MATCHED THEN
INSERT ([Name])
VALUES (i.[Name])
OUTPUT INSERTED.[Id], i._Position;
```

The transaction is gone, as in the single insert case, because MERGE is a single statement protected by an implicit transaction. Also, the temporary table is gone and the OUTPUT clause now sends the generated IDs directly back to the client. This can be four times faster than on EF Core 6, depending on environmental factors such as latency between the application and database.

More about SaveChanges performance

To find out more about SaveChanges performance in EF7, see:

- What's New: Faster SaveChanges
- .NET Blog: Announcing Entity Framework Core 7 Preview 6–Performance Edition
- .NET Data Community Standup Video: Performance Improvements to the EF7 Update Pipeline
- Sample code: SaveChanges performance

Table-per-concrete-type (TPC) inheritance mapping

By default, EF Core maps an inheritance hierarchy of .NET types to a single database table. This is known as the table-per-hierarchy (TPH) mapping strategy. EF Core 5 introduced the table-per-type (TPT) strategy, which supports mapping each .NET type to a different database table. EF7 introduces the table-per-concrete-type (TPC) strategy. TPC also maps .NET types to different tables, but in a way that addresses some common performance issues with the TPT strategy.

The TPC strategy is similar to the TPT strategy except that a different table is created for every concrete type in the hierarchy, but tables are not created for abstract types–hence the name "table-per-concrete-type". As with TPT, the table itself indicates the type of the object saved. However, unlike TPT

mapping, each table contains columns for every property in the concrete type *and its base types*. TPC database schemas are therefore denormalized.

TPC tables

Consider this C# inheritance hierarchy:

```
public abstract class Animal
{
    public int Id { get; set; }
    public string Name { get; set; }
    public abstract string Species { get; }
    public Food? Food { get; set; }
}

public abstract class Pet : Animal
{
    public string? Vet { get; set; }
    public ICollection<Human> Humans { get; } = new List<Human>();
}

public class FarmAnimal : Animal
{
    public override string Species { get; }
    public decimal Value { get; set; }
}

public class Cat : Pet
{
    public string EducationLevel { get; set; }
    public override string Species => "Felis catus";
}

public class Dog : Pet
{
    public string FavoriteToy { get; set; }
    public override string Species => "Canis familiaris";
}
```

```
public class Human : Animal
{
    public override string Species => "Homo sapiens";
    public Animal? FavoriteAnimal { get; set; }
    public ICollection<Pet> Pets { get; } = new List<Pet>();
}
```

This is mapped to TPC tables

using UseTpcMappingStrategy in OnModelCreating:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Animal>().UseTpcMappingStrategy();
}
```

When using SQL Server, the tables created for this hierarchy are:

```
CREATE TABLE [Cats] (
    [Id] int NOT NULL DEFAULT (NEXT VALUE FOR [AnimalSequence]),
    [Name] nvarchar(max) NOT NULL,
    [FoodId] uniqueidentifier NULL,
    [Vet] nvarchar(max) NULL,
    [EducationLevel] nvarchar(max) NOT NULL,
    CONSTRAINT [PK_Cats] PRIMARY KEY ([Id]));

CREATE TABLE [Dogs] (
    [Id] int NOT NULL DEFAULT (NEXT VALUE FOR [AnimalSequence]),
    [Name] nvarchar(max) NOT NULL,
    [FoodId] uniqueidentifier NULL,
    [Vet] nvarchar(max) NULL,
    [FavoriteToy] nvarchar(max) NOT NULL,
    CONSTRAINT [PK_Dogs] PRIMARY KEY ([Id]));

CREATE TABLE [FarmAnimals] (
    [Id] int NOT NULL DEFAULT (NEXT VALUE FOR [AnimalSequence]),
    [Name] nvarchar(max) NOT NULL,
    [FoodId] uniqueidentifier NULL,
```

```
    [Value] decimal(18,2) NOT NULL,
    [Species] nvarchar(max) NOT NULL,
    CONSTRAINT [PK_FarmAnimals] PRIMARY KEY ([Id]));

CREATE TABLE [Humans] (
    [Id] int NOT NULL DEFAULT (NEXT VALUE FOR [AnimalSequence]),
    [Name] nvarchar(max) NOT NULL,
    [FoodId] uniqueidentifier NULL,
    [FavoriteAnimalId] int NULL,
    CONSTRAINT [PK_Humans] PRIMARY KEY ([Id]));
```

TPC queries

TPC generates more efficient queries than TPT, requiring data from fewer tables, and utilizing UNION ALL instead of JOIN. For example, for querying the entire hierarchy, EF7 generates:

```
SELECT [f].[Id], [f].[FoodId], [f].[Name], [f].[Species], [f].[Value], NULL AS [FavoriteAnimalId], NULL
AS [Vet], NULL AS [EducationLevel], NULL AS [FavoriteToy], N'FarmAnimal' AS [Discriminator]
FROM [FarmAnimals] AS [f]
UNION ALL
SELECT [h].[Id], [h].[FoodId], [h].[Name], NULL AS [Species], NULL AS [Value],
[h].[FavoriteAnimalId], NULL AS [Vet], NULL AS [EducationLevel], NULL AS [FavoriteToy], N'Human'
AS [Discriminator]
FROM [Humans] AS [h]
UNION ALL
SELECT [c].[Id], [c].[FoodId], [c].[Name], NULL AS [Species], NULL AS [Value], NULL AS
[FavoriteAnimalId], [c].[Vet], [c].[EducationLevel], NULL AS [FavoriteToy], N'Cat' AS [Discriminator]
FROM [Cats] AS [c]
UNION ALL
SELECT [d].[Id], [d].[FoodId], [d].[Name], NULL AS [Species], NULL AS [Value], NULL AS
[FavoriteAnimalId], [d].[Vet], NULL AS [EducationLevel], [d].[FavoriteToy], N'Dog' AS [Discriminator]
FROM [Dogs] AS [d]
```

This gets even better when querying a subset of types:

```
SELECT [c].[Id], [c].[FoodId], [c].[Name], [c].[Vet], [c].[EducationLevel], NULL AS [FavoriteToy], N'Cat'
AS [Discriminator]
FROM [Cats] AS [c]
```

```
UNION ALL
SELECT [d].[Id], [d].[FoodId], [d].[Name], [d].[Vet], NULL AS [EducationLevel], [d].[FavoriteToy],
N'Dog' AS [Discriminator]
FROM [Dogs] AS [d]
```

And TPC queries really shine when querying for a single leaf type:

```
SELECT [c].[Id], [c].[FoodId], [c].[Name], [c].[Vet], [c].[EducationLevel]
FROM [Cats] AS [c]
```

More about TPC inheritance mapping

To find out more about TPC inheritance mapping in EF7, see:

- What's New: Table-per-concrete-type (TPC) inheritance mapping
- .NET Data Community Standup Video: TPH, TPT, and TPC Inheritance mapping with EF Core
- Sample code: TPC inheritance

Custom Database First templates

EF7 supports T4 templates for customizing the scaffolded code when reverse engineering an EF model from a database. The default templates are added to your project using a dotnet command:

```
dotnet new --install Microsoft.EntityFrameworkCore.Templates
dotnet new ef-templates
```

The templates can then be customized and will automatically be used by dotnet ef dbcontext scaffold and Scaffold-DbContext.

Customizing generated entity types

Let's walk through what it's like to customize a template. By default, EF Core generates the following code for collection navigation properties.

```
public virtual ICollection<Album> Albums { get; } = new List<Album>();
```

Using List<T> is a good default for most applications. However, if you're using a XAML-based framework like WPF, WinUI, or .NET MAUI, you often want to use ObservableCollection<T> to enable data binding.

The EntityType.t4 template can be edited to make this change. For example, the following code is included in the default template:

```
    if (navigation.IsCollection)
    {
#>
    public virtual ICollection<<#= targetType #>> <#= navigation.Name #> { get; } = new List<<#=
targetType #>>();
<#
    }
```

This can easily be changed to use ObservableCollection:

```
public virtual ICollection<<#= targetType #>> <#= navigation.Name #> { get; } = new
ObservableCollection<<#= targetType #>>();
```

Since ObservableCollection is in the System.Collections.ObjectModel namespace, we also need to add a using directive to the scaffolded code:

```
var usings = new List<string>
{
    "System",
    "System.Collections.Generic",
    "System.Collections.ObjectModel"
};
```

More about reverse engineering T4 templates

To find out more about reverse engineering T4 templates in EF7, see:

- What's New: Custom Reverse Engineering Templates
- .NET Data Community Standup Video: Database-first with T4 Templates in EF7–An early look

Custom model-building conventions

EF Core uses a metadata "model" to describe how the application's entity types are mapped to the underlying database. This model is built using a set of around 60 "conventions". The model built by conventions can then be customized using mapping attributes (aka "data annotations") and/or calls to the ModelBuilder API in OnModelCreating.

Starting with EF7, applications can remove or replace any of these conventions, as well as add new conventions. Model building conventions are a powerful way to control the model configuration.

Removing a convention

EF7 allows the default conventions used by EF Core to be removed. For example, it usually makes sense to create indexes for foreign key (FK) columns, and hence there is a built-in convention for this: ForeignKeyIndexConvention. However, indexes add overhead when updating rows, and it may not always be appropriate to create them for all FK columns. To achieve this, the ForeignKeyIndexConvention can be removed when building the model:

```
protected override void ConfigureConventions(ModelConfigurationBuilder configurationBuilder)
{
    configurationBuilder.Conventions.Remove(typeof(ForeignKeyIndexConvention));
}
```

Adding a new convention

A common request from EF Core users is to set a default length for all string properties. This can be accomplished in EF7 by writing a convention:

```
public class MaxStringLengthConvention : IModelFinalizingConvention
{
    private readonly int _maxLength;

    public MaxStringLengthConvention(int maxLength)
    {
        _maxLength = maxLength;
    }

    public void ProcessModelFinalizing(IConventionModelBuilder modelBuilder,
IConventionContext<IConventionModelBuilder> context)
    {
        foreach (var property in modelBuilder.Metadata.GetEntityTypes()
                    .SelectMany(
                        e => e.GetDeclaredProperties()
                            .Where(p => p.ClrType == typeof(string))))
        {
```

```
            property.Builder.HasMaxLength(_maxLength);
        }
    }
}
```

This convention is pretty simple. It finds every string property in the model and sets its max length to the specified value. However, the critical thing about using a convention like this is that properties that have their max length explicitly set using the [MaxLength] or [StringLength] attributes,

or HasMaxLength in OnModelCreating will retain their explicit values. In other words, the default set by the convention is *only used when no other length has been specified*.

This new convention can be used by adding it in ConfigureConventions:

```
protected override void ConfigureConventions(ModelConfigurationBuilder configurationBuilder)
{
    configurationBuilder.Conventions.Add(_ => new MaxStringLengthConvention(256));
}
```

More about custom model-building conventions
To find out more about custom model-building conventions in EF7, see:

- What's New: Model building conventions
- .NET Data Community Standup Video: EF7 Custom Model Conventions
- Sample code: Model building conventions

Stored procedure mapping for insert/update/delete
By default, EF Core generates insert, update, and delete commands that work directly with tables or updatable views. EF7 introduces support for mapping of these commands to stored procedures.

TIP EF Core has always supported querying via stored procedures. The new support in EF7 is specifically about using stored procedures for inserts, updates, and deletes.

Stored procedures are mapped

in OnModelCreating using InsertUsingStoredProcedure, UpdateUsingStoredPr

ocedure, and DeleteUsingStoredProcedure. For example, to map stored procedures for a Person entity type:

```
modelBuilder.Entity<Person>()
    .InsertUsingStoredProcedure(
        "People_Insert",
        storedProcedureBuilder =>
        {
            storedProcedureBuilder.HasParameter(a => a.Name);
            storedProcedureBuilder.HasResultColumn(a => a.Id);
        })
    .UpdateUsingStoredProcedure(
        "People_Update",
        storedProcedureBuilder =>
        {
            storedProcedureBuilder.HasOriginalValueParameter(person => person.Id);
            storedProcedureBuilder.HasOriginalValueParameter(person => person.Name);
            storedProcedureBuilder.HasParameter(person => person.Name);
            storedProcedureBuilder.HasRowsAffectedResultColumn();
        })
    .DeleteUsingStoredProcedure(
        "People_Delete",
        storedProcedureBuilder =>
        {
            storedProcedureBuilder.HasOriginalValueParameter(person => person.Id);
            storedProcedureBuilder.HasOriginalValueParameter(person => person.Name);
            storedProcedureBuilder.HasRowsAffectedResultColumn();
        });
```

This configuration maps to the following stored procedures when using SQL Server:

For inserts

```
CREATE PROCEDURE [dbo].[People_Insert]
    @Name [nvarchar](max)
AS
```

```
BEGIN
    INSERT INTO [People] ([Name])
    OUTPUT INSERTED.[Id]
    VALUES (@Name);
END
```

For updates

```
CREATE PROCEDURE [dbo].[People_Update]
    @Id [int],
    @Name_Original [nvarchar](max),
    @Name [nvarchar](max)
AS
BEGIN
    UPDATE [People] SET [Name] = @Name
    WHERE [Id] = @Id AND [Name] = @Name_Original
    SELECT @@ROWCOUNT
END
```

For deletes

```
CREATE PROCEDURE [dbo].[People_Delete]
    @Id [int],
    @Name_Original [nvarchar](max)
AS
BEGIN
    DELETE FROM [People]
    OUTPUT 1
    WHERE [Id] = @Id AND [Name] = @Name_Original;
END
```

These stored procedures are then used when SaveChangesAsync is called:

```
SET NOCOUNT ON;
EXEC [People_Update] @p1, @p2, @p3;
EXEC [People_Update] @p4, @p5, @p6;
EXEC [People_Delete] @p7, @p8;
EXEC [People_Delete] @p9, @p10;
```

More about stored procedure mapping

To find out more about stored procedure mapping in EF7, see:

- What's New: Stored procedure mapping
- Sample code: Stored procedure mapping

New and improved interceptors and events
EF Core interceptors enable interception, modification, and/or suppression of EF Core operations. EF Core also includes traditional .NET events and logging. EF7 includes the following enhancements to interceptors:

- Interception for creating and populating new entity instances (aka "materialization")
- Interception to modify the LINQ expression tree before a query is compiled
- Interception for optimistic concurrency handling (DbUpdateConcurrencyException)
- Interception for connections before checking if the connection string has been set
- Interception for when EF Core has finished consuming a result set, but before that result set is closed
- Interception for creation of a DbConnection by EF Core
- Interception for DbCommand after it has been initialized

In addition, EF7 includes new traditional .NET events for:

- When an entity is about to be tracked or change state, but before it is actually tracked or change state
- Before and after EF Core detects changes to entities and properties (aka DetectChanges interception)

Materialization interception

The new IMaterializationInterceptor supports interception before and after an entity instance is created, and before and after properties of that instance are initialized. The interceptor can change or replace the entity instance at each point. This allows:

- Setting unmapped properties or calling methods needed for validation, computed values, or flags.
- Using a factory to create instances.
- Creating a different entity instance than EF would normally create, such as an instance from a cache, or of a proxy type.

- Injecting services into an entity instance.

For example, imagine that we want to keep track of the time that an entity was retrieved from the database, perhaps so it can be displayed to a user editing the data. A materialization interceptor can be created for this:

```
public class SetRetrievedInterceptor : IMaterializationInterceptor
{
    public object InitializedInstance(MaterializationInterceptionData materializationData, object
instance)
    {
        if (instance is IHasRetrieved hasRetrieved)
        {
            hasRetrieved.Retrieved = DateTime.UtcNow;
        }

        return instance;
    }
}
```

An instance of this interceptor is registered when configuring the DbContext:

```
public class CustomerContext : DbContext
{
    private static readonly SetRetrievedInterceptor _setRetrievedInterceptor = new();

    public DbSet<Customer> Customers => Set<Customer>();

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        => optionsBuilder
            .AddInterceptors(_setRetrievedInterceptor)
            .UseSqlite("Data Source = customers.db");
}
```

Now, whenever a Customer is queried from the database,

the Retrieved property will be set automatically. For example:

```
await using (var context = new CustomerContext())
```

```
{
    var customer = await context.Customers.SingleAsync(e => e.Name == "Alice");
    Console.WriteLine($"Customer '{customer.Name}' was retrieved at
'{customer.Retrieved.ToLocalTime()}'");
}
```

Produces the following output:

```
Customer 'Alice' was retrieved at '9/22/2022 5:25:54 PM'
```

Lazy initialization of a connection string

Connection strings are often static assets read from a configuration file. These can easily be passed to UseSqlServer or similar when configuring a DbContext. However, the connection string can sometimes change for each context instance. For example, each tenant in a multi-tenant system may have different connection string.

EF7 makes it easier to handle dynamic connections and connection strings through improvements to IDbConnectionInterceptor. This starts with the ability to configure the DbContext without any connection string. For example:

```
services.AddDbContext<CustomerContext>(
    b => b.UseSqlServer();
```

One of the IDbConnectionInterceptor methods can then be implemented to configure the connection before it is used. ConnectionOpeningAsync is a good choice, since it can perform an async operation to obtain the connection string, find an access token, and so on.

```
public class ConnectionStringInitializationInterceptor : DbConnectionInterceptor
{
    private readonly IClientConnectionStringFactory _connectionStringFactory;

    public ConnectionStringInitializationInterceptor(IClientConnectionStringFactory
connectionStringFactory)
    {
        _connectionStringFactory = connectionStringFactory;
    }
```

```
    public override async ValueTask<InterceptionResult> ConnectionOpeningAsync(
        DbConnection connection, ConnectionEventData eventData, InterceptionResult result,
        CancellationToken cancellationToken = new())
    {
        if (string.IsNullOrEmpty(connection.ConnectionString))
        {
            connection.ConnectionString = (await
_connectionStringFactory.GetConnectionStringAsync(cancellationToken));
        }

        return result;
    }
}
```

Note that the connection string is only obtained the first time that a connection is used. After that, the connection string stored on the DbConnection will be used without looking up a new connection string.

Warning Performing an asynchronous lookup for a connection string, access token, or similar every time it is needed can be very slow. Consider caching these things and only refreshing the cached string or token periodically. For example, access tokens can often be used for a significant period of time before needing to be refreshed.

More about interceptors and events

To find out more about interceptors and events in EF7, see:

- What's New: New and improved interceptors and events
- .NET Data Community Standup Video: Intercept this EF7 Preview 6 Event
- Sample code: Materialization interception
- Sample code: Connection string interception

Smaller enhancements

EF7 contains a wealth of smaller new features and enhancements not covered above. Some of these are:

- Query enhancements
    - GroupBy as final operator
    - GroupJoin as final operator

    - [GroupBy entity type]https://learn.microsoft.com/ef/core/what-is-new/ef-core-7.0/whatsnew#groupby-entity-type
    - Subqueries don't reference ungrouped columns from outer query
    - Read-only collections can be used for Contains
    - Translations for aggregate functions
    - Translation of string.IndexOf
    - Translation of GetType for entity types
    - Support for AT TIME ZONE
    - Filtered Include on hidden navigations
    - Cosmos translation for Regex.IsMatch
- DbContext API and behavior enhancements
    - Suppressor for uninitialized DbSet properties
    - Distinguish cancellation from failure in logs
    - New IProperty and INavigation overloads for EntityEntry methods
    - EntityEntry for shared-type entity types
    - ContextInitialized is now logged as Debug
    - IEntityEntryGraphIterator is publicly usable
- Model building enhancements
    - Indexes can be ascending or descending
    - Mapping attribute for composite keys
    - DeleteBehavior mapping attribute
    - Properties mapped to different column names
    - Unidirectional many-to-many relationships
    - Entity splitting
    - SQL Server UTF-8 strings
    - Temporal tables support owned entities
- Improved value generation
    - Value generation for DDD guarded types
    - Sequence-based key generation for SQL Server
- Migrations tooling improvements
    - UseSqlServer etc. accept null
    - Detect when tools are running
- Performance enhancements for proxies
- First-class Windows Forms data binding

Breaking changes
EF7 includes a small number of breaking changes over EF Core 6.

Summary
EF Core 7 (EF7) continues the trend of EF Core releases that make big steps in both performance and functionality. We hope you enjoy using it as much as we have enjoyed creating it. A big thank you from the EF team to everyone who has used and contributed to EF Core over the years.

Welcome to EF7!

EF7 Prerequisites

- EF7 targets .NET 6, which means it can be used on .NET 6 (LTS) or .NET 7.
- EF7 will not run on .NET Framework.

EF7 is the successor to EF Core 6, not to be confused with "classic" EF6. If you are considering upgrading from EF6, please read our guide to port from EF6 to EF Core.

How to get EF7
EF7 is distributed exclusively as a set of NuGet packages. For example, to add the SQL Server provider to your project, you can use the following command using the dotnet tool:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

Installing the EF7 Command Line Interface (CLI)

The dotnet-ef tool must be installed before executing EF7 Core migration or scaffolding commands.

To install the tool globally, use:

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

If you already have the tool installed, you can upgrade it with the following command:

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

The .NET Data Community Standup
The .NET data team is now live streaming every other Wednesday at 10am Pacific Time, 1pm Eastern Time, or 18:00 UTC. Join the stream learn and ask questions about many .NET Data related topics.

- Watch our YouTube playlist of previous shows
- Visit the .NET Community Standup page to preview upcoming shows
- Submit your ideas for a guest, product, demo, or other content to cover

Documentation and Feedback
The starting point for all EF Core documentation is docs.microsoft.com/ef/. Please file issues found and any other feedback on the dotnet/efcore GitHub repo.

Helpful Links
The following links are provided for easy reference and access.

- EF Core Community Standup Playlist: https://aka.ms/efstandups
- Main documentation: https://aka.ms/efdocs
- What's New in EF Core 7: https://aka.ms/efcore-whats-new
- Issues and feature requests for EF Core: https://aka.ms/efcorefeedback
- Entity Framework Roadmap: https://aka.ms/efroadmap
- Bi-weekly updates: https://github.com/dotnet/efcore/issues/27185