New features in EF Core 2.0

Article • 11/10/2020 • 9 minutes to read

.NET Standard 2.0

EF Core now targets .NET Standard 2.0, which means it can work with .NET Core 2.0, .NET Framework 4.6.1, and other libraries that implement .NET Standard 2.0. See Supported .NET Implementations for more details on what is supported.

Modeling

Table splitting

It is now possible to map two or more entity types to the same table where the primary key column(s) will be shared and each row will correspond to two or more entities.

To use table splitting an identifying relationship (where foreign key properties form the primary key) must be configured between all of the entity types sharing the table:

C#

```
modelBuilder.Entity<Product>()
    .HasOne(e => e.Details).WithOne(e => e.Product)
    .HasForeignKey<ProductDetails>(e => e.Id);
modelBuilder.Entity<Product>().ToTable("Products");
modelBuilder.Entity<ProductDetails>().ToTable("Products");
```

Read the section on table splitting for more information on this feature.

Owned types

An owned entity type can share the same .NET type with another owned entity type, but since it cannot be identified just by the .NET type there must be a navigation to it from another entity type. The entity containing the defining navigation is the owner. When querying the owner the owned types will be included by default.

By convention a shadow primary key will be created for the owned type and it will be

mapped to the same table as the owner by using table splitting. This allows to use owned types similarly to how complex types are used in EF6:

C#

```
modelBuilder.Entity<Order>().OwnsOne(p => p.OrderDetails, cb =>
    {
        cb.OwnsOne(c => c.BillingAddress);
        cb.OwnsOne(c => c.ShippingAddress);
    });

public class Order
{
    public int Id { get; set; }
    public OrderDetails OrderDetails { get; set; }
}

public class OrderDetails
{
    public StreetAddress BillingAddress { get; set; }
    public StreetAddress ShippingAddress { get; set; }
}

public class StreetAddress
{
    public string Street { get; set; }
    public string City { get; set; }
}
```

Read the section on owned entity types for more information on this feature.

Model-level query filters

EF Core 2.0 includes a new feature we call Model-level query filters. This feature allows LINQ query predicates (a boolean expression typically passed to the LINQ Where query operator) to be defined directly on Entity Types in the metadata model (usually in OnModelCreating). Such filters are automatically applied to any LINQ queries involving those Entity Types, including Entity Types referenced indirectly, such as through the use of Include or direct navigation property references. Some common applications of this feature are:

- Soft delete - An Entity Types defines an IsDeleted property.
- Multi-tenancy - An Entity Type defines a TenantId property.

Here is a simple example demonstrating the feature for the two scenarios listed above:

C#

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    public int TenantId { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Post>().HasQueryFilter(
            p => !p.IsDeleted
            && p.TenantId == this.TenantId);
    }
}
```

We define a model-level filter that implements multi-tenancy and soft-delete for instances of the `Post` Entity Type. Note the use of a `DbContext` instance-level property: `TenantId`. Model-level filters will use the value from the correct context instance (that is, the context instance that is executing the query).

Filters may be disabled for individual LINQ queries using the IgnoreQueryFilters() operator.

Limitations

- Navigation references are not allowed. This feature may be added based on feedback.
- Filters can only be defined on the root Entity Type of a hierarchy.

Database scalar function mapping

EF Core 2.0 includes an important contribution from Paul Middleton which enables mapping database scalar functions to method stubs so that they can be used in LINQ queries and translated to SQL.

Here is a brief description of how the feature can be used:

Declare a static method on your `DbContext` and annotate it with `DbFunctionAttribute`:

C#

```
public class BloggingContext : DbContext
{
    [DbFunction]
    public static int PostReadCount(int blogId)
    {
        throw new NotImplementedException();
    }
}
```

Methods like this are automatically registered. Once registered, calls to the method in a LINQ query can be translated to function calls in SQL:

C#

```
var query =
    from p in context.Posts
    where BloggingContext.PostReadCount(p.Id) > 5
    select p;
```

A few things to note:

- By convention the name of the method is used as the name of a function (in this case a user-defined function) when generating the SQL, but you can override the name and schema during method registration.
- Currently only scalar functions are supported.
- You must create the mapped function in the database. EF Core migrations will not take care of creating it.

Self-contained type configuration for code first

In EF6 it was possible to encapsulate the code first configuration of a specific entity type by deriving from *EntityTypeConfiguration*. In EF Core 2.0 we are bringing this pattern back:

C#

```
class CustomerConfiguration : IEntityTypeConfiguration<Customer>
{
```

```
    public void Configure(EntityTypeBuilder<Customer> builder)
    {
        builder.HasKey(c => c.AlternateKey);
        builder.Property(c => c.Name).HasMaxLength(200);
    }
}

...
// OnModelCreating
builder.ApplyConfiguration(new CustomerConfiguration());
```

High Performance

DbContext pooling

The basic pattern for using EF Core in an ASP.NET Core application usually involves registering a custom DbContext type into the dependency injection system and later obtaining instances of that type through constructor parameters in controllers. This means a new instance of the DbContext is created for each request.

In version 2.0 we are introducing a new way to register custom DbContext types in dependency injection which transparently introduces a pool of reusable DbContext instances. To use DbContext pooling, use the `AddDbContextPool` instead of `AddDbContext` during service registration:

C#

```
services.AddDbContextPool<BloggingContext>(
    options => options.UseSqlServer(connectionString));
```

If this method is used, at the time a DbContext instance is requested by a controller we will first check if there is an instance available in the pool. Once the request processing finalizes, any state on the instance is reset and the instance is itself returned to the pool.

This is conceptually similar to how connection pooling operates in ADO.NET providers and has the advantage of saving some of the cost of initialization of DbContext instance.

Limitations

The new method introduces a few limitations on what can be done in the `OnConfiguring()` method of the DbContext.

> ⚠ **Warning**
>
> Avoid using DbContext Pooling if you maintain your own state (for example, private fields) in your derived DbContext class that should not be shared across requests. EF Core will only reset the state that it is aware of before adding a DbContext instance to the pool.

Explicitly compiled queries

This is the second opt-in performance feature designed to offer benefits in high-scale scenarios.

Manual or explicitly compiled query APIs have been available in previous versions of EF and also in LINQ to SQL to allow applications to cache the translation of queries so that they can be computed only once and executed many times.

Although in general EF Core can automatically compile and cache queries based on a hashed representation of the query expressions, this mechanism can be used to obtain a small performance gain by bypassing the computation of the hash and the cache lookup, allowing the application to use an already compiled query through the invocation of a delegate.

C#

```
// Create an explicitly compiled query
private static Func<CustomerContext, int, Customer> _customerById =
    EF.CompileQuery((CustomerContext db, int id) =>
        db.Customers
            .Include(c => c.Address)
            .Single(c => c.Id == id));

// Use the compiled query by invoking it
using (var db = new CustomerContext())
{
   var customer = _customerById(db, 147);
}
```

Change Tracking

Attach can track a graph of new and existing entities

EF Core supports automatic generation of key values through a variety of mechanisms. When using this feature, a value is generated if the key property is the CLR default--usually zero or null. This means that a graph of entities can be passed to `DbContext.Attach` or `DbSet.Attach` and EF Core will mark those entities that have a key already set as `Unchanged` while those entities that do not have a key set will be marked as `Added`. This makes it easy to attach a graph of mixed new and existing entities when using generated keys. `DbContext.Update` and `DbSet.Update` work in the same way, except that entities with a key set are marked as `Modified` instead of `Unchanged`.

Query

Improved LINQ translation

Enables more queries to successfully execute, with more logic being evaluated in the database (rather than in-memory) and less data unnecessarily being retrieved from the database.

GroupJoin improvements

This work improves the SQL that is generated for group joins. Group joins are most often a result of sub-queries on optional navigation properties.

String interpolation in FromSql and ExecuteSqlCommand

C# 6 introduced String Interpolation, a feature that allows C# expressions to be directly embedded in string literals, providing a nice way of building strings at runtime. In EF Core 2.0 we added special support for interpolated strings to our two primary APIs that accept raw SQL strings: `FromSql` and `ExecuteSqlCommand`. This new support allows C# string interpolation to be used in a "safe" manner. That is, in a way that protects against common SQL injection mistakes that can occur when dynamically constructing SQL at runtime.

Here is an example:

C#

```
var city = "London";
var contactTitle = "Sales Representative";

using (var context = CreateContext())
{
    context.Set<Customer>()
        .FromSql($@"
            SELECT *
            FROM ""Customers""
            WHERE ""City"" = {city} AND
                ""ContactTitle"" = {contactTitle}")
            .ToArray();
  }
```

In this example, there are two variables embedded in the SQL format string. EF Core will produce the following SQL:

SQL

```
@p0='London' (Size = 4000)
@p1='Sales Representative' (Size = 4000)

SELECT *
FROM ""Customers""
WHERE ""City"" = @p0
    AND ""ContactTitle"" = @p1
```

EF.Functions.Like()

We have added the EF.Functions property which can be used by EF Core or providers to define methods that map to database functions or operators so that those can be invoked in LINQ queries. The first example of such a method is Like():

C#

```
var aCustomers =
    from c in context.Customers
    where EF.Functions.Like(c.Name, "a%")
    select c;
```

Note that Like() comes with an in-memory implementation, which can be handy when working against an in-memory database or when evaluation of the predicate needs to occur on the client side.

Database management

Pluralization hook for DbContext scaffolding

EF Core 2.0 introduces a new *IPluralizer* service that is used to singularize entity type names and pluralize DbSet names. The default implementation is a no-op, so this is just a hook where folks can easily plug in their own pluralizer.

Here is what it looks like for a developer to hook in their own pluralizer:

C#

```
public class MyDesignTimeServices : IDesignTimeServices
{
    public void ConfigureDesignTimeServices(IServiceCollection services)
    {
        services.AddSingleton<IPluralizer, MyPluralizer>();
    }
}

public class MyPluralizer : IPluralizer
{
    public string Pluralize(string name)
    {
        return Inflector.Inflector.Pluralize(name) ?? name;
    }

    public string Singularize(string name)
    {
        return Inflector.Inflector.Singularize(name) ?? name;
    }
}
```

Others

Move ADO.NET SQLite provider to SQLitePCL.raw

This gives us a more robust solution in Microsoft.Data.Sqlite for distributing native SQLite binaries on different platforms.

Only one provider per model

Significantly augments how providers can interact with the model and simplifies how conventions, annotations and fluent APIs work with different providers.

EF Core 2.0 will now build a different IModel for each different provider being used. This is usually transparent to the application. This has facilitated a simplification of lower-level metadata APIs such that any access to *common relational metadata concepts* is always made through a call to `.Relational` instead of `.SqlServer`, `.Sqlite`, etc.

Consolidated logging and diagnostics

Logging (based on ILogger) and Diagnostics (based on DiagnosticSource) mechanisms now share more code.

The event IDs for messages sent to an ILogger have changed in 2.0. The event IDs are now unique across EF Core code. These messages now also follow the standard pattern for structured logging used by, for example, MVC.

Logger categories have also changed. There is now a well-known set of categories accessed through DbLoggerCategory .

DiagnosticSource events now use the same event ID names as the corresponding `ILogger` messages.