Entity Framework Core

Version 5.0
Notes by: rajeeshvengalapurath@gmail.com

DbContext

https://docs.microsoft.com/en-us/ef/core/dbcontext-configuration/

Intro

- ➔ Entity Framework Core does not support multiple parallel operations being run on the same DbContext instance
- ➔ Therefore, always await async calls immediately, or use separate DbContext instances for operations that execute in parallel
- ➔ When EF Core detects an attempt to use a DbContext instance concurrently, you'll see an *InvalidOperationException*
- ➔ The AddDbContext extension method registers DbContext types with a scoped lifetime by default
- ➔ For Blazor Server hosting model, one logical request is used for maintaining the Blazor user circuit, and thus only one scoped DbContext instance is available per user circuit if the default injection scope is used (Use DbContextFactory to fix this issue)

The DbContext lifetime

A DbContext instance is designed to be used for a **single unit-of-work**. This means that the lifetime of a DbContext instance is usually very short.

A Unit of Work

A Unit of Work keeps track of everything you do during a business transaction that can affect the database. When you're done, it figures out everything that needs to be done to alter the database as a result of your work.

A unit-of-work when using Entity Framework Core

- ➔ Creation of a DbContext instance
- ➔ Tracking of entity instances by the context. Entities become tracked by
  - ◆ Being returned from a query
  - ◆ Being added or attached to the context
- ➔ Changes are made to the tracked entities as needed to implement the business rule
- ➔ SaveChanges or SaveChangesAsync is called.
- ➔ The DbContext instance is disposed

Disposal of DbContext

- ➔ It is very important to dispose the DbContext after use. This ensures both that any unmanaged resources are freed

- ➔ DbContext is not thread-safe. Do not share contexts between threads. Make sure to await all async calls before continuing to use the context instance
- ➔ An InvalidOperationException thrown by EF Core code can put the context into an unrecoverable state

DbContext in dependency injection for ASP.NET Core

In many web applications, each HTTP request corresponds to a single unit-of-work. This makes tying the context lifetime to that of the request a good default for web applications

```
public void ConfigureServices(IServiceCollection services)
{
       services.AddDbContext<ApplicationDbContext>(options =>
              options.UseSqlServer("name=ConnectionStrings:DefaultConnection"));
}
```

This registers ApplicationDbContext as a scoped service. The ApplicationDbContext class must expose a public constructor with a DbContextOptions<ApplicationDbContext> parameter

```
public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }
}
```

Constructor injection

```
public class MyController
{
    private readonly ApplicationDbContext _context;

    public MyController(ApplicationDbContext context)
    {
        _context = context;
    }
}
```

Simple DbContext initialization with 'new'

```
public class ApplicationDbContext : DbContext
{
    private readonly string _connectionString;
    public ApplicationDbContext(string connectionString)
    {
        _connectionString = connectionString;
    }
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(_connectionString);
    }
}
```

DbContext factory (e.g. for Blazor) EF Core 5.0

- ➔ Blazor use dependency injection but do not create a service scope that aligns with the desired DbContext lifetime
- ➔ When the application may need to perform multiple units-of-work within this scope
- ➔ Multiple units-of-work within a single HTTP request

In these cases, AddDbContextFactory can be used to register a factory for creation of DbContext instances. See ASP.NET Core Blazor Server with Entity Framework Core (https://docs.microsoft.com/en-us/aspnet/core/blazor/blazor-server-ef-core) for more information on using EF Core with Blazor.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContextFactory<ApplicationDbContext>(
        options =>
            options.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=Test"));
}

public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }
}
```

Constructor injection

```
private readonly IDbContextFactory<ApplicationDbContext> _contextFactory;

public MyController(IDbContextFactory<ApplicationDbContext> contextFactory)
{
    _contextFactory = contextFactory;
}
```

The injected factory can then be used to construct DbContext instances in the service code

```
public void DoSomething()
{
    using (var context = _contextFactory.CreateDbContext())
    {
        ...
    }
}
```

Notice that the DbContext instances created in this way are not managed by the application's service provider and therefore must be disposed by the application

Unit of work with DbContextFactory

```
public class UnitOfWork<T> : IUnitOfWork
    where T : System.Data.Entity.DbContext, new()
{
    private readonly IDbContextFactory _dbContextFactory;
    private T _dbContext;
```

```
    public UnitOfWork(IDbContextFactory dbContextFactory)
    {
        _dbContextFactory = dbContextFactory;
    }

    public T DbContext => _dbContext ??
       (_dbContext =_dbContextFactory.GetDbContext<T>());

    public void Commit()
    {
        DbContext.SaveChanges();
    }
}
```

Avoiding DbContext threading issues

- ➔ Entity Framework Core does not support multiple parallel operations being run on the same DbContext instance
- ➔ Therefore, always await async calls immediately, or use separate DbContext instances for operations that execute in parallel
- ➔ Asynchronous methods enable EF Core to initiate operations that access the database in a non-blocking way.
- ➔ If async caller a caller does not await the completion of one of these methods, and proceeds to perform other operations on the DbContext, the state of the DbContext can be, (and very likely will be) corrupted

Implicitly sharing DbContext instances via dependency injection

- ➔ The AddDbContext extension method registers DbContext types with a scoped lifetime by default
- ➔ This is safe from concurrent access issues in most ASP.NET Core applications because there is only one thread executing each client request at a given time
- ➔ For Blazor Server hosting model, one logical request is used for maintaining the Blazor user circuit, and thus only one scoped DbContext instance is available per user circuit if the default injection scope is used (Use DbContextFactory to fix this issue)

Creating and configuring a model

https://docs.microsoft.com/en-us/ef/core/modeling/

The model

Entity classes + Context object

A model is made up of entity classes and a context object that represents a session with the database. EF supports the following model development approaches:

- ➔ Generate a model from an existing database.
- ➔ Hand code a model to match the database.
- ➔ Once a model is created, use EF Migrations to create a database from the model

Querying

```
using (var db = new BloggingContext())
{
    var blogs = db.Blogs
        .Where(b => b.Rating > 3)
        .OrderBy(b => b.Url)
        .ToList();
}
```

Saving data

```
using (var db = new BloggingContext())
{
    var blog = new Blog { Url = "http://sample.com" };
    db.Blogs.Add(blog);
    db.SaveChanges();
}
```

Fluent API to configure a model

➔ Fluent API configuration **has the highest precedence** and will override conventions and data annotations.

```
internal class MyContext : DbContext
{
       …
       protected override void OnModelCreating(ModelBuilder modelBuilder)
       {
              modelBuilder.Entity<Blog>()
                     .Property(b => b.Url)
                     .IsRequired();
       }
}
```

Grouping configuration

Method 1

```
public class BlogEntityTypeConfiguration : IEntityTypeConfiguration<Blog>
{
    public void Configure(EntityTypeBuilder<Blog> builder)
    {
        builder
            .Property(b => b.Url)
            .IsRequired();
    }
}
```

Then just invoke the Configure method from OnModelCreating

```
new BlogEntityTypeConfiguration().Configure(modelBuilder.Entity<Blog>());
```

Method 2

It is possible to apply all configuration specified in types implementing IEntityTypeConfiguration in a given assembly.

```
modelBuilder.ApplyConfigurationsFromAssembly(typeof(BlogEntityTypeConfiguration).As
sembly);
```

Data annotations to configure a model

```
public class Blog
{
       public int BlogId { get; set; }

       [Required]
       public string Url { get; set; }
}
```

Entity Types

Including types in the model

In the code sample below, all types are included:

- ➔ Blog is included because it's *exposed in a DbSet property* on the context.
- ➔ Post is included because it's *discovered via the Blog.Posts navigation property*.
- ➔ AuditEntry because it is *specified in OnModelCreating*

```
internal class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<AuditEntry>();
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    ...
}

public class AuditEntry
{
    ...
}
```

Excluding types from the model

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Ignore<BlogMetadata>();
}
```

Excluding from migrations

It is sometimes useful to have the same entity type mapped in multiple DbContext types. The ability to exclude tables from migrations was introduced in EF Core 5.0.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<IdentityUser>()
        .ToTable("AspNetUsers", t => t.ExcludeFromMigrations());
}
```

With this configuration migrations will not create the AspNetUsers table, but IdentityUser is still included in the model and can be used normally.

Table name

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .ToTable("blogs");
}
```

Table schema

For example, Microsoft SQL Server will use the dbo schema

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .ToTable("blogs", schema: "blogging");
}
```

Table-valued function mapping

It's possible to map an entity type to a table-valued function (TVF) instead of a table in the database. In order to map an entity to a table-valued function the function must be parameterless

```
modelBuilder.Entity<BlogWithMultiplePosts>().HasNoKey().ToFunction("BlogsWithMultip
lePosts");
```

Query:

```
var query = from b in context.Set<BlogWithMultiplePosts>()
            where b.PostCount > 3
            select new { b.Url, b.PostCount };
```

Included and excluded properties

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
```

```
    modelBuilder.Entity<Blog>()
        .Ignore(b => b.LoadedFromDatabase);
}
```

Column names

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.BlogId)
        .HasColumnName("blog_id");
}
```

Column data types

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>(
        eb =>
        {
            eb.Property(b => b.Url).HasColumnType("varchar(200)");
            eb.Property(b => b.Rating).HasColumnType("decimal(5, 2)");
        });
}
```

Maximum length

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .HasMaxLength(500);
}
```

Required and optional properties

- ➔ By convention, a property whose .NET type can contain null will be configured as optional, whereas properties whose .NET type cannot contain null will be configured as required
- ➔ C# 8 introduced a new feature called nullable reference types (NRT), which allows reference types to be annotated, indicating whether it is valid for them to contain null or not
- ➔ If nullable reference types are disabled (the default), all properties with .NET reference types are configured as optional by convention (for example, string).
- ➔ If nullable reference types are enabled, properties will be configured based on the C# nullability of their .NET type: string? will be configured as optional, but string will be configured as required.

Without NRT (default)

```
public class CustomerWithoutNullableReferenceTypes
{
    public int Id { get; set; }

    [Required] // Data annotations needed to configure as required
    public string FirstName { get; set; }

    [Required]
```

```
    public string LastName { get; set; } // Data annotations needed to configure as
required

    public string MiddleName { get; set; } // Optional by convention
}
```

Witht NRT

```
public class Customer
{
    public int Id { get; set; }
    public string FirstName { get; set; } // Required by convention
    public string LastName { get; set; } // Required by convention
    public string? MiddleName { get; set; } // Optional by convention

    // Note the following use of constructor binding, which avoids compiled
warnings
    // for uninitialized non-nullable properties.
    public Customer(string firstName, string lastName, string? middleName = null)
    {
        FirstName = firstName;
        LastName = lastName;
        MiddleName = middleName;
    }
}
```

Explicit configuration

A property that would be optional by convention can be configured to be required as follows:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .IsRequired();
}
```

Keys

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Car>()
        .HasKey(c => c.LicensePlate);
}
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Car>()
        .HasKey(c => new { c.State, c.LicensePlate });
}
```

Primary key name

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .HasKey(b => b.BlogId)
        .HasName("PrimaryKey_BlogId");
}
```

Generated Values

Default values

If a row is inserted without a value for that column, the default value will be used

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Rating)
        .HasDefaultValue(3);
}
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Created)
        .HasDefaultValueSql("getdate()"); //With SQL fragment
}
```

Computed columns

```
modelBuilder.Entity<Person>()
    .Property(p => p.DisplayName)
    .HasComputedColumnSql("[LastName] + ', ' + [FirstName]"); //with an expression
```

👆 The above creates a virtual computed column, whose value is computed every time it is fetched from the database

```
modelBuilder.Entity<Person>()
    .Property(p => p.NameLength)
    .HasComputedColumnSql("LEN([LastName]) + LEN([FirstName])", stored: true);
```

👆 Above, Specify that a computed column be stored (sometimes called persisted), meaning that it is computed on every update of the row, and is stored on disk alongside regular columns. (Stored computed columns was added in EF Core 5.0)

Explicitly configuring value generation

EF Core automatically sets up value generation for primary keys. For non-key properties:

```
modelBuilder.Entity<Blog>()
       .Property(b => b.Inserted)
       .ValueGeneratedOnAdd();

modelBuilder.Entity<Blog>()
        .Property(b => b.LastUpdated)
        .ValueGeneratedOnAddOrUpdate();
```

We are not specifying how the values are to be generated; that depends on the database provider being used

Date/time value generation

Creation timestamp

```
modelBuilder.Entity<Blog>()
        .Property(b => b.Created)
        .HasDefaultValueSql("getdate()");
```

Be sure to select the appropriate function, as several may exist (e.g. GETDATE() vs. GETUTCDATE()).

Overriding value generation

Although a property is configured for value generation, in many cases you may still explicitly specify a value for it

```
modelBuilder.Entity<Blog>().Property(b => b.LastUpdated)
        .ValueGeneratedOnAddOrUpdate()
        .Metadata.SetAfterSaveBehavior(PropertySaveBehavior.Save);
```

No value generation

For example, a primary key of type int is usually implicitly configured as value-generated-on-add (e.g. identity column on SQL Server). You can disable this via the following

```
modelBuilder.Entity<Blog>()
        .Property(b => b.BlogId)
        .ValueGeneratedNever();
```

Concurrency Tokens

- ➔ Database concurrency refers to situations in which multiple processes or users access or change the same data in a database at the same time
- ➔ The situation when another user has performed an operation that conflicts with the current operation is known as concurrency conflict
- ➔ Concurrency control refers to specific mechanisms used to ensure data consistency in presence of concurrent changes

How concurrency control works in EF Core

- ➔ Properties configured as concurrency tokens are used to implement optimistic concurrency control
- ➔ Whenever an update or delete operation is performed during SaveChanges, the value of the concurrency token on the database is compared against the original value read by EF Core
  - ◆ If the values match, the operation can complete
  - ◆ If the values do not match, EF Core assumes that another user has performed a conflicting operation and aborts the current transaction
- ➔ On relational databases EF Core includes a check for the value of the concurrency token in the WHERE clause of any UPDATE or DELETE statements. After executing the statements, EF Core reads the number of rows that were affected

```
modelBuilder.Entity<Person>()
        .Property(p => p.LastName)
```

```
        .IsConcurrencyToken(); //No db changes detected on migration
```

Timestamp/rowversion

A timestamp/rowversion is a property for which a new value is automatically generated by the database every time a row is inserted or updated. The property is also treated as a concurrency token, ensuring that you get an exception if a row you are updating has changed since you queried it

```
modelBuilder.Entity<Blog>()
            .Property(p => p.Timestamp)
            .IsRowVersion();

public class Blog
{
    ...
    public byte[] Timestamp { get; set; }
}
```

Shadow and Indexer Properties

Shadow properties are properties that aren't defined in your .NET entity class but are defined for that entity type in the EF Core model

```
public class Post
{
    ..

    // Since there is no CLR property which holds the foreign
    // key for this relationship, a shadow property is created.
    public Blog Blog { get; set; }
}
```

Configuring shadow properties

```
internal class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property<DateTime>("LastUpdated");
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Accessing shadow properties

Shadow properties cannot be accessed after a no-tracking query since the entities returned are not tracked by the change tracker

Shadow property values can be obtained and changed through the ChangeTracker API:

```
context.Entry(myBlog).Property("LastUpdated").CurrentValue = DateTime.Now;
```

Shadow properties can be referenced in LINQ queries via the EF.Property static method:

```
var blogs = context.Blogs
    .OrderBy(b => EF.Property<DateTime>(b, "LastUpdated"));
```

Configuring indexer properties

Indexer properties are entity type properties, which are backed by an indexer in .NET entity class. They can be accessed using the indexer on the .NET class instances. It also allows you to add additional properties to the entity type without changing the CLR class

```
modelBuilder.Entity<Blog>().IndexerProperty<DateTime>("LastUpdated");
```

Relationships

- ➔ Dependent entity: This is the entity that contains the foreign key properties. Sometimes referred to as the 'child' of the relationship.
- ➔ Principal entity: This is the entity that contains the primary/alternate key properties. Sometimes referred to as the 'parent' of the relationship.
- ➔ Principal key: The properties that uniquely identify the principal entity. This may be the primary key or an alternate key.
- ➔ Foreign key: The properties in the dependent entity that are used to store the principal key values for the related entity.
- ➔ Navigation property: A property defined on the principal and/or dependent entity that references the related entity.
    - ◆ Collection navigation property: A navigation property that contains references to many related entities.
    - ◆ Reference navigation property: A navigation property that holds a reference to a single related entity.
    - ◆ Inverse navigation property: When discussing a particular navigation property, this term refers to the navigation property on the other end of the relationship.
- ➔ Self-referencing relationship: A relationship in which the dependent and the principal entity types are the same.

Cascade delete

- ➔ Cascade means dependent entities are also deleted
- ➔ By convention, cascade delete will be set to Cascade for required relationships and ClientSetNull for optional relationships

Manual configuration

```
modelBuilder.Entity<Post>()
            .HasOne(p => p.Blog)
            .WithMany(b => b.Posts);
```

Single navigation property

If you only have one navigation property then there are parameterless overloads of WithOne and WithMany

```
modelBuilder.Entity<Blog>()
            .HasMany(b => b.Posts)
            .WithOne();
```

Configuring navigation properties

After the navigation property has been created, you may need to further configure it. This feature was introduced in EF Core 5.0.

```
modelBuilder.Entity<Blog>()
        .HasMany(b => b.Posts)
        .WithOne();

    modelBuilder.Entity<Blog>()
        .Navigation(b => b.Posts)
        .UsePropertyAccessMode(PropertyAccessMode.Property);
```

Foreign key

```
modelBuilder.Entity<Post>()
            .HasOne(p => p.Blog)
            .WithMany(b => b.Posts)
            .HasForeignKey(p => p.BlogForeignKey);
```

Foreign key constraint name

```
modelBuilder.Entity<Post>()
        .HasOne(p => p.Blog)
        .WithMany(b => b.Posts)
        .HasForeignKey(p => p.BlogId)
        .HasConstraintName("ForeignKey_Post_Blog");
```

Without navigation property

```
modelBuilder.Entity<Post>()
            .HasOne<Blog>()
            .WithMany()
            .HasForeignKey(p => p.BlogId);

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public int BlogId { get; set; }
}
```

Principal key

If you want the foreign key to reference a property other than the primary key. The property that you configure as the principal key will automatically be set up as an alternate key

```
modelBuilder.Entity<RecordOfSale>()
            .HasOne(s => s.Car)
            .WithMany(c => c.SaleHistory)
            .HasForeignKey(s => s.CarLicensePlate)
            .HasPrincipalKey(c => c.LicensePlate);
```

Required and optional relationships

Required navigation expects the related entity to always be present. If necessary related entity is filtered out by the query filter, the parent entity *wouldn't be in result* either. So you may get fewer elements than expected in result

Caution: Using required navigation to access entity which has global query filter defined may lead to unexpected results.

```
modelBuilder.Entity<Post>()
        .HasOne(p => p.Blog)
        .WithMany(b => b.Posts)
        .IsRequired();
```

Cascade delete

```
modelBuilder.Entity<Post>()
        .HasOne(p => p.Blog)
        .WithMany(b => b.Posts)
        .OnDelete(DeleteBehavior.Cascade);
```

When cascading behaviors happen

Cascading deletes are needed when a dependent/child entity can no longer be associated with its current principal/parent

One-to-one

```
modelBuilder.Entity<Blog>()
            .HasOne(b => b.BlogImage)
            .WithOne(i => i.Blog)
            .HasForeignKey<BlogImage>(b => b.BlogForeignKey);
```

Many-to-many

Many to many relationships require a collection navigation property on both sides

```
public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public ICollection<Tag> Tags { get; set; }
}
```

```
public class Tag
{
    public string TagId { get; set; }

    public ICollection<Post> Posts { get; set; }
}
```

Joining relationships configuration ?

```
modelBuilder
    .Entity<Post>()
    .HasMany(p => p.Tags)
    .WithMany(p => p.Posts)
    .UsingEntity(j => j.ToTable("PostTags")); //?
```

Indirect many-to-many relationships

```
modelBuilder.Entity<PostTag>()
            .HasOne(pt => pt.Post)
            .WithMany(p => p.PostTags)
            .HasForeignKey(pt => pt.PostId);

        modelBuilder.Entity<PostTag>()
            .HasOne(pt => pt.Tag)
            .WithMany(t => t.PostTags)
            .HasForeignKey(pt => pt.TagId);
```

Indexes

➔ By convention, an index is created in each property (or set of properties) that are used as a foreign key

Sample

```
modelBuilder.Entity<Blog>()
        .HasIndex(b => b.Url);
```

Composite index

```
modelBuilder.Entity<Person>()
        .HasIndex(p => new { p.FirstName, p.LastName });
```

Index uniqueness

```
modelBuilder.Entity<Blog>()
        .HasIndex(b => b.Url)
        .IsUnique();
```

Index name

```
modelBuilder.Entity<Blog>()
        .HasIndex(b => b.Url)
        .HasDatabaseName("Index_Url");
```

Index filter

```
modelBuilder.Entity<Blog>()
        .HasIndex(b => b.Url)
        .IsUnique()
        .HasFilter(null);
```

Included columns

Some relational databases allow you to configure a set of columns which get included in the index, but aren't part of its "key". This can significantly improve query performance when all columns in the query are included in the index either as key or nonkey columns

```
modelBuilder.Entity<Post>()
        .HasIndex(p => p.Url)
        .IncludeProperties(
            p => new { p.Title, p.PublishedOn });
```

Inheritance

Entity type hierarchy mapping

EF will not automatically scan for base or derived types; this means that if you want a CLR type in your hierarchy to be mapped, you must explicitly specify that type on your model.

The following sample exposes a DbSet for Blog and its subclass RssBlog. If Blog has any other subclass, it will not be included in the model

```
internal class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<RssBlog> RssBlogs { get; set; }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}

public class RssBlog : Blog
{
    public string RssUrl { get; set; }
}
```

Table-per-hierarchy and discriminator configuration

By default, EF maps the inheritance using the table-per-hierarchy (TPH) pattern. TPH uses a single table to store the data for all types in the hierarchy, and a discriminator column (to be configured) is used to identify which type each row represents

To added the discriminator explicitly:

```
modelBuilder.Entity<Blog>()
        .HasDiscriminator<string>("blog_type") //new column to identify class
```

```
        .HasValue<Blog>("blog_base")
        .HasValue<RssBlog>("blog_rss");
```

To added the discriminator implicitly:

```
modelBuilder.Entity<Blog>()
        .Property("Discriminator") // "Discriminator" is a name to new column
        .HasMaxLength(200);
```

Also discriminator can also be mapped to a regular .NET property in your entity

```
modelBuilder.Entity<Blog>()
        .HasDiscriminator(b => b.BlogType);
```

Table-per-type configuration

In the TPT mapping pattern, all the types are mapped to individual tables. EF Core 5.0.

Warning: In many cases, TPT shows inferior performance when compared to TPH

```
modelBuilder.Entity<Blog>().ToTable("Blogs");
modelBuilder.Entity<RssBlog>().ToTable("RssBlogs");
```

Sequences

A sequence generates unique, sequential numeric values in the database. Sequences are not associated with a specific table, and multiple tables can be set up to draw values from the same sequence.

```
modelBuilder.HasSequence<int>("OrderNumbers");

    modelBuilder.Entity<Order>()
        .Property(o => o.OrderNo)
        .HasDefaultValueSql("NEXT VALUE FOR shared.OrderNumbers");
```

Note that the specific SQL used to generate a value from a sequence is database-specific

Configuring sequence settings

```
modelBuilder.HasSequence<int>("OrderNumbers", schema: "shared")
        .StartsAt(1000)
        .IncrementsBy(5);
```

Value Conversions

Value converters allow property values to be converted when reading from or writing to the database. More at https://docs.microsoft.com/en-us/ef/core/modeling/value-conversions

```
public class Rider
{
    public int Id { get; set; }
    public EquineBeast Mount { get; set; }
}

public enum EquineBeast
```

```
{
    Donkey,
    Mule,
    Horse,
    Unicorn
}

modelBuilder
        .Entity<Rider>()
        .Property(e => e.Mount)
        .HasConversion(
            v => v.ToString(),
            v => (EquineBeast)Enum.Parse(typeof(EquineBeast), v));
```

Pre-defined conversions

```
modelBuilder
        .Entity<Rider>()
        .Property(e => e.Mount)
        .HasConversion<string>(); //Read and write as string automatically
```

or

```
modelBuilder
    .Entity<Rider2>()
    .Property(e => e.Mount)
    .HasColumnType("nvarchar(24)");
```

The ValueConverter class

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    var converter = new ValueConverter<EquineBeast, string>(
        v => v.ToString(),
        v => (EquineBeast)Enum.Parse(typeof(EquineBeast), v));

    modelBuilder
        .Entity<Rider>()
        .Property(e => e.Mount)
        .HasConversion(converter);
}
```

Built-in converters

For example, using .HasConversion<int>() on a bool

```
modelBuilder
        .Entity<User>()
        .Property(e => e.IsActive)
        .HasConversion<int>();
```

creating an instance of the built-in BoolToZeroOneConverter<TProvider> and setting it explicitly

```
modelBuilder
        .Entity<User>()
        .Property(e => e.IsActive)
        .HasConversion(converter);
```

Column facets and mapping hints

```
modelBuilder
        .Entity<Rider>()
        .Property(e => e.Mount)
        .HasConversion<string>()
        .HasMaxLength(20)
        .IsUnicode(false);
```

or

```
var converter = new ValueConverter<EquineBeast, string>(
        v => v.ToString(),
        v => (EquineBeast)Enum.Parse(typeof(EquineBeast), v));

    modelBuilder
        .Entity<Rider>()
        .Property(e => e.Mount)
        .HasConversion(converter)
        .HasMaxLength(20)
        .IsUnicode(false);
```

Data Seeding

➔ Data seeding is the process of populating a database with an initial set of data

Model seed data

EF Core migrations can automatically compute what insert, update or delete operations need to be applied when upgrading the database to a new version of the model

```
modelBuilder.Entity<Blog>().HasData(new Blog { BlogId = 1, Url = "..." });
```

To add entities that have a relationship the foreign key values need to be specified:

```
modelBuilder.Entity<Post>().HasData(
    new Post { BlogId = 1, PostId = 1, Title = "First post", Content = "Test 1" });
```

Owned entity types can be seeded in a similar fashion:

```
modelBuilder.Entity<Post>().OwnsOne(p => p.AuthorName).HasData(
    new { PostId = 1, First = "Andriy", Last = "Svyryd" },
    new { PostId = 2, First = "Diego", Last = "Vega" });
```

Alternatively, you can use `context.Database.EnsureCreated()` to create a new database containing the seed data

Custom initialization logic

A straightforward and powerful way to perform data seeding is to use DbContext.SaveChanges() before the main application logic begins execution.

Warning: The seeding code should not be part of the normal app execution as this can cause concurrency issues when multiple instances are running and would also require the app having permission to modify the database schema

```
using (var context = new DataSeedingContext())
{
    context.Database.EnsureCreated();

    var testBlog = context.Blogs.FirstOrDefault(b => b.Url == "http://test.com");
    if (testBlog == null)
    {
        context.Blogs.Add(new Blog { Url = "http://test.com" });
    }

    context.SaveChanges();
}
```

Entity types with constructors

Binding to mapped properties

If EF Core finds a parameterized constructor with parameter names and types that match those of mapped properties, then it will call the parameterized constructor with values for those properties and will not set each property explicitly

```
public class Blog
{
    public Blog(int id, string name)
    {
        Id = id;
        Name = name;
        //Author = author;
    }

    public int Id { get; set; }

    public string Name { get; set; }
    public string Author { get; set; }

    public ICollection<Post> Posts { get; } = new List<Post>();
}
```

Injecting services

```
public class Blog
{
    ..

    private Blog(BloggingContext context)
    {
        Context = context;
    }

    private BloggingContext Context { get; set; }

    ...
}
```

Table Splitting

```
public class Order
{
    public int Id { get; set; }
    public OrderStatus? Status { get; set; }
    public DetailedOrder DetailedOrder { get; set; }
}
public class DetailedOrder
{
    public int Id { get; set; }
    public OrderStatus? Status { get; set; }
    public string BillingAddress { get; set; }
    public string ShippingAddress { get; set; }
    public byte[] Version { get; set; }
}

modelBuilder.Entity<DetailedOrder>(
    dob =>
    {
        dob.ToTable("Orders");
        dob.Property(o => o.Status).HasColumnName("Status");
    });

modelBuilder.Entity<Order>(
    ob =>
    {
        ob.ToTable("Orders");
        ob.Property(o => o.Status).HasColumnName("Status");
        ob.HasOne(o => o.DetailedOrder).WithOne()
            .HasForeignKey<DetailedOrder>(o => o.Id);
    });
```

Owned Entity Types

EF Core allows you to model entity types that can only ever appear on navigation properties of other entity types. More at https://docs.microsoft.com/en-us/ef/core/modeling/owned-entities

Explicit configuration

```
[Owned]
public class StreetAddress
{
    public string Street { get; set; }
    public string City { get; set; }
}
public class Order
{
    public int Id { get; set; }
    public StreetAddress ShippingAddress { get; set; }
}
```

Will create table Order only and not StreetAddress:

```
Order (Table)
       Id
       ShippingAddress_Street
```

```
        ShippingAddress_City
```

Migrations

```
Add-Migration InitialCreate
Update-Database
```

Excluding parts of your model

```
modelBuilder.Entity<IdentityUser>()
        .ToTable("AspNetUsers", t => t.ExcludeFromMigrations());
```

Adding raw SQL

Add this to generated migrations file

```
migrationBuilder.AddColumn<string>(
    name: "FullName",
    table: "Customer",
    nullable: true);

migrationBuilder.Sql(
@"
    UPDATE Customer
    SET FullName = FirstName + ' ' + LastName;
");

migrationBuilder.DropColumn(
    name: "FirstName",
    table: "Customer");
```

Arbitrary changes via raw SQL

Raw SQL can also be used to manage database objects that EF Core isn't aware of. To do this, add a migration without making any model change; an empty migration will be generated, which you can then populate with raw SQL operations

```
migrationBuilder.Sql(
@"
    EXEC ('CREATE PROCEDURE getFullName
        @LastName nvarchar(50),
        @FirstName nvarchar(50)
    AS
        RETURN @LastName + @FirstName;')");
```

This can be used to manage any aspect of your database, including:

- ➔ Stored procedures
- ➔ Full-Text Search
- ➔ Functions
- ➔ Triggers
- ➔ Views

Remove a migration

```
Remove-Migration
```

Warning: Avoid removing any migrations which have already been applied to production databases. Doing so means you won't be able to revert those migrations from the databases, and may break the assumptions made by subsequent migrations

Listing migrations

```
Get-Migration
```

Resetting all migrations

It's also possible to reset all migrations and create a single one without losing your data. This is sometimes called "squashing", and involves some manual work:

- ➔ Delete your Migrations folder
- ➔ Create a new migration and generate a SQL script for it
- ➔ In your database, delete all rows from the migrations history table
- ➔ Insert a single row into the migrations history, to record that the first migration has already been applied, since your tables are already there. The insert SQL is the last operation in the SQL script generated above.

Warning: Any custom migration code will be lost when the Migrations folder is deleted

Applying Migrations (Deployment)

SQL scripts

The recommended way to deploy migrations to a production database is by generating SQL scripts. The advantages of this strategy include the following:

- ➔ SQL scripts can be reviewed for accuracy; this is important since applying schema changes to production databases is a potentially dangerous operation that could involve data loss.
- ➔ In some cases, the scripts can be tuned to fit the specific needs of a production database.
- ➔ SQL scripts can be used in conjunction with a deployment technology, and can even be generated as part of your CI process.
- ➔ SQL scripts can be provided to a DBA, and can be managed and archived separately.

1. The following generates a SQL script from a blank database to the latest migration:

```
Script-Migration
```

2. The following generates a SQL script from the given migration to the latest migration. (From to)

```
Script-Migration AddNewTables AddAuditTable
```

Apply migrations at runtime

It's possible for the application itself to apply migrations programmatically. This approach is inappropriate for managing production databases

```
public static void Main(string[] args)
{
    var host = CreateHostBuilder(args).Build();

    using (var scope = host.Services.CreateScope())
    {
        var db = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();
        db.Database.Migrate();
    }

    host.Run();
}
```

Migrations in Team Environments

When you merge migrations from your teammates, you may get conflicts in your model snapshot file. If both changes are unrelated, the merge is trivial and the two migrations can coexist

```
<<<<<<< Mine
b.Property<bool>("Deactivated");
=======
b.Property<int>("LoyaltyPoints");
>>>>>>> Theirs
```

To

```
b.Property<bool>("Deactivated");
b.Property<int>("LoyaltyPoints");
```

Resolving conflicts

Sometimes you encounter a true conflict when merging the model snapshot model. For example, you and your teammate may each have renamed the same property. resolve it by re-creating your migration. Follow these steps:

- ➔ Abort the merge and rollback to your working directory before the merge
- ➔ Remove your migration (but keep your model changes)
- ➔ Merge your teammate's changes into your working directory
- ➔ Re-add your migration

Using a Separate Migrations Project

See: https://github.com/RajeeshVengalapurath/EntityFrameworkStudy

You may want to store your migrations in a different project than the one containing your DbContext. You can also use this strategy to maintain multiple sets of migrations, for example, one for development and another for release-to-release upgrades

- ➔ Create a new class library.
- ➔ Add a reference to your DbContext project.
- ➔ Move the migrations and model snapshot files to the class library.
- ➔ Configure the migrations assembly:

```
services.AddDbContext<ApplicationDbContext>(
       options =>
              options.UseSqlServer(
              Configuration.GetConnectionString("DefaultConnection"),
              x => x.MigrationsAssembly("WebApplication1.Migrations")));
```

- ➔ Add a reference to your migrations project from the startup project
- ➔ If this causes a circular dependency, you can update the base output path of the migrations project instead ?

Custom Migrations History Table

```
protected override void OnConfiguring(DbContextOptionsBuilder options)
    => options.UseSqlServer(
        _connectionString,
        x => x.MigrationsHistoryTable("__MyMigrationsHistory", "mySchema"));
```

Querying Data

Entity Framework Core uses Language-Integrated Query (LINQ) to query data from the database

Writing performant queries

See “Query null semantics” section

- ➔ Comparing non-nullable columns is simpler and faster than comparing nullable columns. Consider marking columns as non-nullable whenever possible.
- ➔ Checking for equality (==) is simpler and faster than checking for non-equality (!=), because query doesn't need to distinguish between null and false result. Use equality comparison whenever possible. However, simply negating == comparison is effectively the same as !=, so it doesn't result in performance improvement.
- ➔ In some cases, it is possible to simplify a complex comparison by filtering out null values from a column explicitly - for example when no null values are present or these values are not relevant in the result

Loading all data

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs.ToList();
}
```

Loading a single entity

```
var blog = context.Blogs
        .Single(b => b.BlogId == 1);
```

Filtering

```
var blogs = context.Blogs
        .Where(b => b.Url.Contains("dotnet"))
        .ToList();
```

Client evaluation in the top-level projection

```
var blogs = context.Blogs
    .OrderByDescending(blog => blog.Rating)
    .Select(
        blog => new { Id = blog.BlogId, Url = StandardizeUrl(blog.Url) })
    .ToList();
```

All other aspects of the query are evaluated in the database, but passing the returned URL through the helper method `StandardizeUrl` is done on the client

Unsupported client evaluation

```
var blogs = context.Blogs
    .Where(blog => StandardizeUrl(blog.Url).Contains("dotnet"))
    .ToList();
```

Because the filter can't be applied in the database, all the data needs to be pulled into memory to apply the filter on the client. Based on the filter and the amount of data on the server, client evaluation could result in poor performance. So Entity Framework Core blocks such client evaluation and throws a runtime exception.

Explicit client evaluation

You may need to force into client evaluation explicitly in certain cases like following

- ➔ The amount of data is small so that evaluating on the client doesn't incur a huge performance penalty.
- ➔ The LINQ operator being used has no server-side translation

In such cases, you can explicitly opt into client evaluation by calling methods like AsEnumerable or ToList (AsAsyncEnumerable or ToListAsync for async)

```
var blogs = context.Blogs
    .AsEnumerable()
    .Where(blog => StandardizeUrl(blog.Url).Contains("dotnet"))
    .ToList();
```

Tracking vs. No-Tracking Queries

If an entity is tracked, any changes detected in the entity will be persisted to the database during SaveChanges(). **Note:** Keyless entity types are never tracked

Tracking queries

By default, queries that return entity types are tracking. In the following example, the change to the blogs rating will be detected and persisted to the database during SaveChanges().

```
var blog = context.Blogs.SingleOrDefault(b => b.BlogId == 1);
blog.Rating = 5;
context.SaveChanges();
```

No-tracking queries

- ➔ No tracking queries are useful when the results are used in a read-only scenario
- ➔ They're quicker to execute because there's no need to set up the change tracking information
- ➔ If you don't need to update the entities retrieved from the database, then a no-tracking query should be used

```
var blogs = context.Blogs
    .AsNoTracking()
    .ToList();
```

To change the default tracking behavior at the context instance level

```
context.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;
var blogs = context.Blogs.ToList();
```

Identity resolution

The default behaviour for AsNoTracking() is not to perform identity resolution. This means if your query returns 4 books and their authors, you'll get 8 instances in total even if 2 books have the same author. In Entity Framework Core 5.0 you can use AsNoTrackingWithIdentityResolution() to not track entities but still perform identity resolution so you won't have duplicate instances

```
var blogs = context.Blogs
    .AsNoTrackingWithIdentityResolution()
    .ToList();
```

Tracking and custom projections

👇 In the following query, which returns an anonymous type, the instances of Blog in the result set will be tracked

```
var blog = context.Blogs
    .Select(
        b =>
            new { Blog = b, PostCount = b.Posts.Count() });
```

👇 If the result set contains entity types coming out from LINQ composition, EF Core will track them:

```
var blog = context.Blogs
    .Select(
        b =>
            new { Blog = b, Post = b.Posts.OrderBy(p => p.Rating).LastOrDefault()
});
```

👇 If the result set doesn't contain any entity types, then no tracking is done

```
var blog = context.Blogs
    .Select(
        b =>
            new { Id = b.BlogId, b.Url });
```

👇 If EF Core materializes an entity instance for client evaluation, it will be tracked. Here, since we're passing blog entities to the client method StandardizeURL, EF Core will track the blog instances too

```
var blogs = context.Blogs
```

```
    .OrderByDescending(blog => blog.Rating)
    .Select(
        blog => new { Id = blog.BlogId, Url = StandardizeUrl(blog) })
    .ToList();
```

Loading Related Data

Eager loading

You can use the Include method to specify related data to be included in query results

```
var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .ToList();

var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .Include(blog => blog.Owner)
        .ToList();
```

Including multiple levels

```
var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .ThenInclude(post => post.Author)
        .ToList();

var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .ThenInclude(post => post.Author)
        .ThenInclude(author => author.Photo)
        .ToList();

using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .ThenInclude(post => post.Author)
        .ThenInclude(author => author.Photo)
        .Include(blog => blog.Owner)
        .ThenInclude(owner => owner.Photo)
        .ToList();
}
```

👇 You may want to include multiple related entities for one of the entities that is being included. For example, when querying Blogs, you include Posts and then want to include both the Author and Tags of the Posts. To include both, you need to specify each include path starting at the root

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .ThenInclude(post => post.Author)
        .Include(blog => blog.Posts)
        .ThenInclude(post => post.Tags)
        .ToList();
}
```

👇 You can also load multiple navigations using a single Include method. This is possible for navigation "chains" that are all references, or when they end with a single collection

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Owner.AuthoredPosts)
        .ThenInclude(post => post.Blog.Owner.Photo)
        .ToList();
}
```

Filtered include

This feature was introduced in EF Core 5.0. When applying Include to load related data, you can add certain enumerable operations to the included collection navigation, which allows for filtering and sorting of the results. Supported operations are: Where, OrderBy, OrderByDescending, ThenBy, ThenByDescending, Skip, and Take

```
var filteredBlogs = context.Blogs
        .Include(
            blog => blog.Posts
                .Where(post => post.BlogId == 1)
                .OrderByDescending(post => post.Title)
                .Take(5))
        .ToList();

var filteredBlogs = context.Blogs
        .Include(blog => blog.Posts.Where(post => post.BlogId == 1))
        .ThenInclude(post => post.Author)
        .Include(blog => blog.Posts)
        .ThenInclude(post => post.Tags.OrderBy(postTag => postTag.TagId).Skip(3))
        .ToList();

var filteredBlogs = context.Blogs
        .Include(blog => blog.Posts.Where(post => post.BlogId == 1))
        .ThenInclude(post => post.Author)
        .Include(blog => blog.Posts.Where(post => post.BlogId == 1))
        .ThenInclude(post => post.Tags.OrderBy(postTag => postTag.TagId).Skip(3))
        .ToList();
```

Caution:

- ➔ In case of tracking queries, results of Filtered Include may be unexpected due to navigation fixup
- ➔ All relevant entities that have been queried for previously and have been stored in the Change Tracker will be present in the results of Filtered Include query, even if they don't meet the requirements of the filter
- ➔ Consider using NoTracking queries or re-create the DbContext when using Filtered Include in those situations

Include on derived types

```
public class SchoolContext : DbContext
{
    public DbSet<Person> People { get; set; }
```

```
    public DbSet<School> Schools { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<School>().HasMany(s => s.Students).WithOne(s =>
s.School);
    }
}

public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
}

public class Student : Person
{
    public School School { get; set; }
}

public class School
{
    public int Id { get; set; }
    public string Name { get; set; }

    public List<Student> Students { get; set; }
}
```

Contents of School navigation of all People who are Students can be eagerly loaded using a number of patterns

Using cast

```
context.People.Include(person => ((Student)person).School).ToList()
```

Using as operator

```
context.People.Include(person => (person as Student).School).ToList()
```

Using overload of Include that takes parameter of type string

```
context.People.Include("School").ToList()
```

Explicit Loading

```
var blog = context.Blogs
        .Single(b => b.BlogId == 1);

    context.Entry(blog)
        .Collection(b => b.Posts)
        .Load();

    context.Entry(blog)
        .Reference(b => b.Owner)
        .Load(); //Load is a void function
```

Querying related entities

```
var postCount = context.Entry(blog)
        .Collection(b => b.Posts)
        .Query()
```

```
        .Count();
```

Filter:

```
var blog = context.Blogs
        .Single(b => b.BlogId == 1);

var goodPosts = context.Entry(blog)
        .Collection(b => b.Posts)
        .Query()
        .Where(p => p.Rating > 3)
        .ToList();
```

Lazy Loading

Warning: Lazy loading can cause unneeded extra database round trips to occur (the so-called N+1 problem), and care should be taken to avoid this. See the performance section for more details.

Lazy loading with proxies

The simplest way to use lazy-loading is by installing the Microsoft.EntityFrameworkCore.Proxies package and enabling it with a call to UseLazyLoadingProxies

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    => optionsBuilder
        .UseLazyLoadingProxies()
        .UseSqlServer(myConnectionString);

Or

.AddDbContext<BloggingContext>(
    b => b.UseLazyLoadingProxies()
          .UseSqlServer(myConnectionString));
```

Lazy loading without proxies

https://docs.microsoft.com/en-us/ef/core/querying/related-data/lazy#lazy-loading-without-proxies

Lazy-loading proxies work by injecting the ILazyLoader service into an entity, as described in Entity Type Constructors

Related data and serialization

Because EF Core automatically does fix-up of navigation properties, you can end up with cycles in your object graph. For example, loading a blog and its related posts will result in a blog object that references a collection of posts. Each of those posts will have a reference back to the blog

Some serialization frameworks don't allow such cycles. For example, Json.NET will throw an exception if a cycle is found

You can configure Json.NET to ignore cycles that it finds in the object graph

```
services.AddMvc()
        .AddJsonOptions(
            options => options.SerializerSettings.ReferenceLoopHandling =
                           Newtonsoft.Json.ReferenceLoopHandling.Ignore
        );
```

Split queries

In relational databases, all related entities are loaded by introducing JOINs in a single query. If a typical blog has multiple related posts, rows for these posts will duplicate the blog's information. This duplication leads to the so-called "cartesian explosion" problem. EF Core uses single query mode by default in the absence of any configuration

Note: One-to-one related entities are always loaded via JOINs in the same query, as it has no performance impact

```
SELECT [b].[BlogId], [b].[OwnerId], [b].[Rating], [b].[Url], [p].[PostId],
[p].[AuthorId], [p].[BlogId], [p].[Content], [p].[Rating], [p].[Title]
FROM [Blogs] AS [b]
LEFT JOIN [Post] AS [p] ON [b].[BlogId] = [p].[BlogId]
ORDER BY [b].[BlogId], [p].[PostId]
```

EF allows you to specify that a given LINQ query should be split into multiple SQL queries. Instead of JOINs, split queries generate an additional SQL query for each included collection navigation.

Note: This feature was introduced in EF Core 5.0. It only works when using Include

```
var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .AsSplitQuery()
        .ToList();
```

It will produce the following SQL:

```
SELECT [b].[BlogId], [b].[OwnerId], [b].[Rating], [b].[Url]
FROM [Blogs] AS [b]
ORDER BY [b].[BlogId]

SELECT [p].[PostId], [p].[AuthorId], [p].[BlogId], [p].[Content], [p].[Rating],
[p].[Title], [b].[BlogId]
FROM [Blogs] AS [b]
INNER JOIN [Post] AS [p] ON [b].[BlogId] = [p].[BlogId]
ORDER BY [b].[BlogId]
```

Enabling split queries globally

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .UseSqlServer(
            @"Server=(localdb)...",
            o => o.UseQuerySplittingBehavior(QuerySplittingBehavior.SplitQuery));
}
```

When split queries are configured as the default, it's still possible to configure specific queries to execute as single queries

```
var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .AsSingleQuery()
        .ToList();
```

Drawbacks

- ➔ While most databases guarantee data consistency for single queries, no such guarantees exist for multiple queries.
- ➔ Each query currently implies an additional network roundtrip to your database. Multiple network roundtrips can degrade performance, especially where latency to the database is high (for example, cloud services).
- ➔ While some databases allow consuming the results of multiple queries at the same time (SQL Server with MARS, Sqlite), most allow only a single query to be active at any given point. So all results from earlier queries must be buffered in your application's memory before executing later queries, which leads to increased memory requirements.

Complex Query Operators

Join

```
var query = from photo in context.Set<PersonPhoto>()
            join person in context.Set<Person>()
                on photo.PersonPhotoId equals person.PhotoId
            select new { person, photo };

SELECT [p].[PersonId], [p].[Name], [p].[PhotoId], [p0].[PersonPhotoId],
[p0].[Caption], [p0].[Photo]
FROM [PersonPhoto] AS [p0]
INNER JOIN [Person] AS [p] ON [p0].[PersonPhotoId] = [p].[PhotoId]
```

If the key selectors are anonymous types, EF Core generates a join condition to compare equality component-wise:

```
var query = from photo in context.Set<PersonPhoto>()
            join person in context.Set<Person>()
                on new { Id = (int?)photo.PersonPhotoId, photo.Caption }
                equals new { Id = person.PhotoId, Caption = "SN" }
            select new { person, photo };

SELECT [p].[PersonId], [p].[Name], [p].[PhotoId], [p0].[PersonPhotoId],
[p0].[Caption], [p0].[Photo]
FROM [PersonPhoto] AS [p0]
INNER JOIN [Person] AS [p] ON ([p0].[PersonPhotoId] = [p].[PhotoId] AND
([p0].[Caption] = N'SN'))
```

GroupJoin

Executing a query like the following example generates a result of Blog & IEnumerable<Post>

```
var query = from b in context.Set<Blog>()
            join p in context.Set<Post>()
                on b.BlogId equals p.PostId into grouping
            select new { b, grouping };

var query = from b in context.Set<Blog>()
            join p in context.Set<Post>()
                on b.BlogId equals p.PostId into grouping
            select new { b, Posts = grouping.Where(p =>
p.Content.Contains("EF")).ToList() };
```

SelectMany

- ➔ In a way, it's a join but without any condition so every outer element is connected with an element from the collection source
- ➔ Depending on how the collection selector is related to the outer data source, SelectMany can translate into various different queries on the server side

Cross Join

```
var query = from b in context.Set<Blog>()
            from p in context.Set<Post>()
            select new { b, p };

SELECT [b].[BlogId], [b].[OwnerId], [b].[Rating], [b].[Url], [p].[PostId],
[p].[AuthorId], [p].[BlogId], [p].[Content], [p].[Rating], [p].[Title]
FROM [Blogs] AS [b]
CROSS JOIN [Posts] AS [p]
```

Inner Join

```
var query = from b in context.Set<Blog>()
            from p in context.Set<Post>().Where(p => b.BlogId == p.BlogId)
            select new { b, p };

SELECT [b].[BlogId], [b].[OwnerId], [b].[Rating], [b].[Url], [p].[PostId],
[p].[AuthorId], [p].[BlogId], [p].[Content], [p].[Rating], [p].[Title]
FROM [Blogs] AS [b]
INNER JOIN [Posts] AS [p] ON [b].[BlogId] = [p].[BlogId]
```

Left Join

```
var query2 = from b in context.Set<Blog>()
             from p in context.Set<Post>().Where(p => b.BlogId ==
p.BlogId).DefaultIfEmpty()
             select new { b, p };

SELECT [b].[BlogId], [b].[OwnerId], [b].[Rating], [b].[Url], [p].[PostId],
[p].[AuthorId], [p].[BlogId], [p].[Content], [p].[Rating], [p].[Title]
FROM [Blogs] AS [b]
LEFT JOIN [Posts] AS [p] ON [b].[BlogId] = [p].[BlogId]
```

GroupBy

```
var query = from p in context.Set<Post>()
            group p by p.AuthorId
            into g
            select new { g.Key, Count = g.Count() };

SELECT [p].[AuthorId] AS [Key], COUNT(*) AS [Count]
FROM [Posts] AS [p]
GROUP BY [p].[AuthorId]

var query = from p in context.Set<Post>()
            group p by p.AuthorId
            into g
            where g.Count() > 0
            orderby g.Key
            select new { g.Key, Count = g.Count() };
```

```
SELECT [p].[AuthorId] AS [Key], COUNT(*) AS [Count]
FROM [Posts] AS [p]
GROUP BY [p].[AuthorId]
HAVING COUNT(*) > 0
ORDER BY [p].[AuthorId]
```

The aggregate operators EF Core supports are as follows

- ➔ Average
- ➔ Count
- ➔ LongCount
- ➔ Max
- ➔ Min
- ➔ Sum

Left Join

While Left Join isn't a LINQ operator, relational databases have the concept of a Left Join which is frequently used in queries. A particular pattern in LINQ queries gives the same result as a LEFT JOIN on the server

```
var query = from b in context.Set<Blog>()
            join p in context.Set<Post>()
                on b.BlogId equals p.BlogId into grouping
            from p in grouping.DefaultIfEmpty()
            select new { b, p };
```

```
SELECT [b].[BlogId], [b].[OwnerId], [b].[Rating], [b].[Url], [p].[PostId],
[p].[AuthorId], [p].[BlogId], [p].[Content], [p].[Rating], [p].[Title]
FROM [Blogs] AS [b]
LEFT JOIN [Posts] AS [p] ON [b].[BlogId] = [p].[BlogId]
```

Raw SQL Queries

Basic raw SQL queries

```
var blogs = context.Blogs
    .FromSqlRaw("SELECT * FROM dbo.Blogs")
    .ToList();
Raw SQL queries can be used to execute a stored procedure.

var blogs = context.Blogs
    .FromSqlRaw("EXECUTE dbo.GetMostPopularBlogs")
    .ToList();
```

Passing parameters

Warning: Always use parameterization for raw SQL queries

When introducing any user-provided values into a raw SQL query, care must be taken to avoid SQL injection attacks. In addition to validating that such values don't contain invalid characters, always use parameterization which sends the values separate from the SQL text.

In particular, never pass a concatenated or interpolated string ($"") with non-validated user-provided values into FromSqlRaw or ExecuteSqlRaw. The FromSqlInterpolated and ExecuteSqlInterpolated methods allow using string interpolation syntax in a way that protects against SQL injection attacks.

```
var user = "johndoe";

var blogs = context.Blogs
    .FromSqlRaw("EXECUTE dbo.GetMostPopularBlogsForUser {0}", user)
    .ToList();
```

DbParameter

```
var user = new SqlParameter("user", "johndoe");

var blogs = context.Blogs
    .FromSqlRaw("EXECUTE dbo.GetMostPopularBlogsForUser @user", user)
    .ToList();

Or

var user = new SqlParameter("user", "johndoe");

var blogs = context.Blogs
    .FromSqlRaw("EXECUTE dbo.GetMostPopularBlogsForUser @filterByUser=@user", user)
    .ToList();
```

Composing with LINQ

You can compose on top of the initial raw SQL query using LINQ operators. EF Core will treat it as subquery and compose over it in the database

```
var searchTerm = "Lorem ipsum";

var blogs = context.Blogs
    .FromSqlInterpolated($"SELECT * FROM dbo.SearchBlogs({searchTerm})")
    .Where(b => b.Rating > 3)
    .OrderByDescending(b => b.Rating)
    .ToList();

SELECT [b].[BlogId], [b].[OwnerId], [b].[Rating], [b].[Url]
FROM (
    SELECT * FROM dbo.SearchBlogs(@p0)
) AS [b]
WHERE [b].[Rating] > 3
ORDER BY [b].[Rating] DESC
```

Including related data

```
var searchTerm = "Lorem ipsum";

var blogs = context.Blogs
    .FromSqlInterpolated($"SELECT * FROM dbo.SearchBlogs({searchTerm})")
    .Include(b => b.Posts)
    .ToList();
```

Change Tracking

```
var searchTerm = "Lorem ipsum";
```

```
var blogs = context.Blogs
    .FromSqlInterpolated($"SELECT * FROM dbo.SearchBlogs({searchTerm})")
    .AsNoTracking()
    .ToList();
```

Limitations

- ➔ The SQL query must return data for all properties of the entity type.
- ➔ The column names in the result set must match the column names that properties are mapped to.
- ➔ The SQL query can't contain related data. However, in many cases you can compose on top of the query using the Include operator to return related data (see Including related data).

User-defined function mapping

```
public class AppDbContext : DbContext
{
       //If OnModelCreating is not called mapping fails
       protected override void OnModelCreating(ModelBuilder modelBuilder)
       {
              ...
              modelBuilder.HasDbFunction(typeof(AppDbContext)
                     .GetMethod(nameof(PostCountForBlog), new[] { typeof(int) }))
                     .HasName("PostCountForBlog").HasSchema("dbo");
       }

       //This function can be outside dbcontext
       public int PostCountForBlog(int id) => throw new NotSupportedException();
}
```

❌ The below code didn't work, coz OnModelCreating is not triggered

```
int count = _dbContext.PostCountForBlog(1);
```

Can be used like this:

```
var blog = _dbContext.Blogs.Where(b => _dbContext.PostCountForBlog(1)>1);
```

Configuring nullability of user-defined function based on its arguments

```
modelBuilder.HasDbFunction(
    typeof(BloggingContext).GetMethod(nameof(ConcatStringsOptimized), new[] {
typeof(string), typeof(string) }),
    b =>
    {
        b.HasName("ConcatStrings");
        b.HasParameter("prm1").PropagatesNullability();
        b.HasParameter("prm2").PropagatesNullability();
    });
```

Global Query Filters

The predicate expressions passed to the HasQueryFilter calls will now automatically be applied to any LINQ queries for those types. Some common applications of this feature are

- ➔ Soft delete - An Entity Type defines an IsDeleted property.

➔ Multi-tenancy - An Entity Type defines a TenantId property.

```
public class Blog
{
       #pragma warning disable IDE0051, CS0169 // Remove unused private members
        private string _tenantId; //??
       #pragma warning restore IDE0051, CS0169 // Remove unused private members

        public int BlogId { get; set; }
        ...

        public List<Post> Posts { get; set; }
}

public class Post
{
        public int PostId { get; set; }
        ...
        public bool IsDeleted { get; set; }

        public Blog Blog { get; set; }
}
```

Note the declaration of a _tenantId field on the Blog entity. This field will be used to associate each Blog instance with a specific tenant

```
modelBuilder.Entity<Blog>()
       .HasQueryFilter(b => EF.Property<string>(b, "_tenantId") == _tenantId); //?
modelBuilder.Entity<Post>().HasQueryFilter(p => !p.IsDeleted);
```

Use of navigations

```
modelBuilder.Entity<Blog>().HasMany(b => b.Posts).WithOne(p => p.Blog);
modelBuilder.Entity<Blog>().HasQueryFilter(b => b.Posts.Count > 0);
modelBuilder.Entity<Post>().HasQueryFilter(p => p.Title.Contains("fish"));
```

Accessing entity with query filter using required navigation

Required navigation expects the related entity to always be present. If necessary related entity is filtered out by the query filter, the parent entity wouldn't be in result either. So you may get fewer elements than expected in result

Caution: Using required navigation to access entity which has global query filter defined may lead to unexpected results.

```
modelBuilder.Entity<Blog>().HasMany(b => b.Posts).WithOne(p => p.Blog)
       .IsRequired();
modelBuilder.Entity<Blog>().HasQueryFilter(b => b.Url.Contains("fish"));
```

Disabling Filters

```
blogs = db.Blogs
    .Include(b => b.Posts)
    .IgnoreQueryFilters()
    .ToList();
```

Limitations

Filters can only be defined for the root Entity Type of an inheritance hierarchy.

Query tags

Query tags help correlate LINQ queries in code with generated SQL queries captured in logs

```
var myLocation = new Point(1, 2);
var nearestPeople = (from f in context.People.TagWith("This is my spatial query!")
                     orderby f.Location.Distance(myLocation) descending
                     select f).Take(5).ToList();

-- This is my spatial query!

SELECT TOP(@__p_1) [p].[Id], [p].[Location]
FROM [People] AS [p]
ORDER BY [p].[Location].STDistance(@__myLocation_0) DESC
```

Query null semantics

SQL databases operate on 3-valued logic (true, false, null) when performing comparisons, as opposed to the boolean logic of C#. When translating LINQ queries to SQL, EF Core tries to compensate for the difference by introducing additional null checks for some elements of the query

```
public class NullSemanticsEntity
{
    public int Id { get; set; }
    public int Int { get; set; }
    public int? NullableInt { get; set; }
    public string String1 { get; set; }
    public string String2 { get; set; }
}

var query1 = context.Entities.Where(e => e.Id == e.Int);
var query2 = context.Entities.Where(e => e.Id == e.NullableInt);
var query3 = context.Entities.Where(e => e.Id != e.NullableInt);
var query4 = context.Entities.Where(e => e.String1 == e.String2);
var query5 = context.Entities.Where(e => e.String1 != e.String2);
```

The first two queries produce simple comparisons. In the first query, both columns are non-nullable so null checks are not needed. In the second query, NullableInt could contain null, but Id is non-nullable; comparing null to non-null yields null as a result, which would be filtered out by WHERE operation. So no additional terms are needed either.

```
SELECT [e].[Id], [e].[Int], [e].[NullableInt], [e].[String1], [e].[String2]
FROM [Entities] AS [e]
WHERE [e].[Id] = [e].[Int]

SELECT [e].[Id], [e].[Int], [e].[NullableInt], [e].[String1], [e].[String2]
FROM [Entities] AS [e]
WHERE [e].[Id] = [e].[NullableInt]
```

The third query introduces a null check. When NullableInt is null the comparison Id <> NullableInt yields null, which would be filtered out by WHERE operation. However, from the boolean logic

perspective this case should be returned as part of the result. Hence EF Core adds the necessary check to ensure that

```
SELECT [e].[Id], [e].[Int], [e].[NullableInt], [e].[String1], [e].[String2]
FROM [Entities] AS [e]
WHERE ([e].[Id] <> [e].[NullableInt]) OR [e].[NullableInt] IS NULL
```

Queries four and five show the pattern when both columns are nullable. It's worth noting that the <> operation produces a more complicated (and potentially slower) query than the == operation.

```
SELECT [e].[Id], [e].[Int], [e].[NullableInt], [e].[String1], [e].[String2]
FROM [Entities] AS [e]
WHERE ([e].[String1] = [e].[String2]) OR ([e].[String1] IS NULL AND [e].[String2]
IS NULL)

SELECT [e].[Id], [e].[Int], [e].[NullableInt], [e].[String1], [e].[String2]
FROM [Entities] AS [e]
WHERE (([e].[String1] <> [e].[String2]) OR ([e].[String1] IS NULL OR [e].[String2]
IS NULL)) AND ([e].[String1] IS NOT NULL OR [e].[String2] IS NOT NULL)
```

Treatment of nullable values in functions

Many functions in SQL can only return a null result if some of their arguments are null. EF Core takes advantage of this to produce more efficient queries. The query below illustrates the optimization

```
var query = context.Entities.Where(e => e.String1.Substring(0, e.String2.Length) ==
null);
```

The generated SQL is as follows (we don't need to evaluate the SUBSTRING function since it will be only null when either of the arguments to it is null.):

```
SELECT [e].[Id], [e].[Int], [e].[NullableInt], [e].[String1], [e].[String2]
FROM [Entities] AS [e]
WHERE [e].[String1] IS NULL OR [e].[String2] IS NULL
```

Writing performant queries

- ➔ Comparing non-nullable columns is simpler and faster than comparing nullable columns. Consider marking columns as non-nullable whenever possible.
- ➔ Checking for equality (==) is simpler and faster than checking for non-equality (!=), because query doesn't need to distinguish between null and false result. Use equality comparison whenever possible. However, simply negating == comparison is effectively the same as !=, so it doesn't result in performance improvement.
- ➔ In some cases, it is possible to simplify a complex comparison by filtering out null values from a column explicitly - for example when no null values are present or these values are not relevant in the result

Saving Data

Each context instance has a ChangeTracker that is responsible for keeping track of changes that need to be written to the database. As you make changes to instances of your entity classes, these changes are recorded in the ChangeTracker and then written to the database when you call SaveChanges. The database provider is responsible for translating the changes into

database-specific operations (for example, INSERT, UPDATE, and DELETE commands for a relational database).

For most database providers, SaveChanges is transactional. This means all the operations will either succeed or fail and the operations will never be left partially applied.

Adding Data

```
using (var context = new BloggingContext())
{
    var blog = new Blog { Url = "http://example.com" };
    context.Blogs.Add(blog);
    context.SaveChanges();
}
```

Updating Data

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.First();
    blog.Url = "http://example.com/blog";
    context.SaveChanges();
}
```

Deleting Data

If the entity already exists in the database, it will be deleted during SaveChanges. If the entity has not yet been saved to the database (that is, it is tracked as added) then it will be removed from the context and will no longer be inserted when SaveChanges is called.

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.First();
    context.Blogs.Remove(blog);
    context.SaveChanges();
}
```

Saving Related Data

Adding a graph of new entities

If you create several new related entities, adding one of them to the context will cause the others to be added too

```
using (var context = new BloggingContext())
{
    var blog = new Blog
    {
        Url = "http://blogs.msdn.com/dotnet",
        Posts = new List<Post>
        {
            new Post { Title = "Intro to C#" },
            new Post { Title = "Intro to VB.NET" },
        }
```

```
    };

    context.Blogs.Add(blog);
    context.SaveChanges();
}
```

Tip: ? Use the EntityEntry.State property to set the state of just a single entity. For example, `context.Entry(blog).State = EntityState.Modified.`

Adding a related entity

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Include(b => b.Posts).First();
    var post = new Post { Title = "Intro to EF Core" };

    blog.Posts.Add(post);
    context.SaveChanges();
}
```

Changing relationships

The post entity is updated to belong to the new blog entity because its Blog navigation property is set to point to blog. Note that blog will also be inserted into the database because it is a new entity that is referenced by the navigation property of an entity that is already tracked by the context (post).

```
using (var context = new BloggingContext())
{
    var blog = new Blog { Url = "http://blogs.msdn.com/visualstudio" };
    var post = context.Posts.First();

    post.Blog = blog;
    context.SaveChanges();
}
```

Removing relationships

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Include(b => b.Posts).First();
    var post = blog.Posts.First();

    blog.Posts.Remove(post);
    context.SaveChanges();
}
```

Cascade Delete

If the principal/parent entity is deleted, then the foreign key values of the dependents/children will no longer match the primary or alternate key of any principal/parent. This is an invalid state, and will cause a referential constraint violation in most databases.

There are two options to avoid this referential constraint violation:

- ➔ Set the FK values to null (for optional relationships where the foreign key property (and the database column to which it is mapped) must be nullable)

➔ Also delete the dependent/child entities (for any kind of relationship and is known as "cascade delete".)

When cascading behaviors happen

Cascading deletes are needed when a dependent/child entity can no longer be associated with its current principal/parent

```
var blog = context.Blogs.OrderBy(e => e.Name).Include(e => e.Posts).First();
context.Remove(blog);
context.SaveChanges();
```

Severing a relationship

Rather than deleting the blog, we could instead sever the relationship between each post and its blog

```
var blog = context.Blogs.OrderBy(e => e.Name).Include(e => e.Posts).First();

foreach (var post in blog.Posts)
{
    post.Blog = null;
}

context.SaveChanges();
```

OR

```
var blog = context.Blogs.OrderBy(e => e.Name).Include(e => e.Posts).First();
blog.Posts.Clear();
context.SaveChanges();
```

Cascade delete in the database

```
CREATE TABLE [Posts] (
    [Id] int NOT NULL IDENTITY,
    [BlogId] int NOT NULL,
    CONSTRAINT [PK_Posts] PRIMARY KEY ([Id]),
    CONSTRAINT [FK_Posts_Blogs_BlogId] FOREIGN KEY ([BlogId]) REFERENCES [Blogs]
([Id]) ON DELETE CASCADE
);
```

If we know that the database is configured like this, then we can delete a blog without first loading posts and the database will take care of deleting all the posts that were related to that blog

```
var blog = context.Blogs.OrderBy(e => e.Name).First();
context.Remove(blog);
context.SaveChanges();
```

Notice that there is no Include for posts, so they are not loaded

Cascading nulls

Optional relationships have nullable foreign key properties mapped to nullable database columns. This means that the foreign key value can be set to null when the current principal/parent is deleted or is severed from the dependent/child. Likewise, if the relationship is severed

Configuring cascading behaviors

```
modelBuilder
        .Entity<Blog>()
        .HasOne(e => e.Owner)
        .WithOne(e => e.OwnedBlog)
        .OnDelete(DeleteBehavior.ClientCascade);
```

DeleteBehavior	Impact on database schema
Cascade	ON DELETE CASCADE
Restrict	ON DELETE NO ACTION
NoAction	database default
SetNull	ON DELETE SET NULL
ClientSetNull	ON DELETE NO ACTION
ClientCascade	ON DELETE NO ACTION
ClientNoAction	database default

Transactions

By default, if the database provider supports transactions, all changes in a single call to SaveChanges are applied in a transaction. If any of the changes fail, then the transaction is rolled back and none of the changes are applied to the database. This means that SaveChanges is guaranteed to either completely succeed, or leave the database unmodified if an error occurs

Controlling transactions

A master detail save scenario

```
using (var trans = _dbContext.Database.BeginTransaction())
{
       try
       {
              Blog blog = new Blog { Url = "Url12" };

              _dbContext.Blogs.Add(blog);
              _dbContext.SaveChanges();

              _dbContext.Posts.Add(
                     new Post { BlogId = blog.Id, Title = "test", Content = "test"
              });
              _dbContext.SaveChanges();

              trans.Commit();
       }
       catch (Exception ex)
       {
              trans.Rollback();
       }
}
```

Savepoints

EF Core 5.0. When SaveChanges is invoked and a transaction is already in progress on the context, EF automatically creates a savepoint before saving any data. Savepoints are points within a database transaction which may later be rolled back to, if an error occurs or for any other reason

It's also possible to manually manage savepoints

```
try
{
    context.Blogs.Add(new Blog { Url = "https://devblogs.microsoft.com/dotnet/" });
    context.SaveChanges();

    transaction.CreateSavepoint("BeforeMoreBlogs");

    context.Blogs.Add(new Blog { Url = "https://devblog.." });
    context.Blogs.Add(new Blog { Url = "https://devblogs.microsoft.com/aspnet/" });
    context.SaveChanges();

    transaction.Commit();
}
catch (Exception)
{
       // If a failure occurred,
       we rollback to the savepoint and can continue the transaction
       transaction.RollbackToSavepoint("BeforeMoreBlogs");

       // TODO: Handle failure, possibly retry inserting blogs
}
```

Cross-context transaction

You can also share a transaction across multiple context instances. Sharing a DbConnection requires the ability to pass a connection into a context when constructing it

```
using var connection = new SqlConnection(connectionString);
var options = new DbContextOptionsBuilder<BloggingContext>()
    .UseSqlServer(connection)
    .Options;

using var context1 = new BloggingContext(options);
using var transaction = context1.Database.BeginTransaction();
try
{
    context1.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });
    context1.SaveChanges();

    using (var context2 = new BloggingContext(options))
    {
        context2.Database.UseTransaction(transaction.GetDbTransaction());

        var blogs = context2.Blogs
            .OrderBy(b => b.Url)
            .ToList();
    }

    transaction.Commit();
}
catch (Exception)
{
    // TODO: Handle failure
}
```

Using external DbTransactions

```
using var connection = new SqlConnection(connectionString);
connection.Open();

using var transaction = connection.BeginTransaction();

try
{
    var command = connection.CreateCommand();
    command.Transaction = transaction;
    command.CommandText = "DELETE FROM dbo.Blogs";
    command.ExecuteNonQuery();

    var options = new DbContextOptionsBuilder<BloggingContext>()
        .UseSqlServer(connection)
        .Options;

    using (var context = new BloggingContext(options))
    {
        context.Database.UseTransaction(transaction);
        context.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });
        context.SaveChanges();
    }

    transaction.Commit();
}
catch (Exception)
{
    // TODO: Handle failure
}
```

Disconnected entities

A DbContext instance will automatically track entities returned from the database. However, sometimes entities are queried using one context instance and then saved using a different instance. This often happens in "disconnected" scenarios such as a web application where the entities are queried, sent to the client, modified, sent back to the server in a request, and then saved. In this case, the second context instance needs to know whether the entities are new (should be inserted) or existing (should be updated).

Identifying new entities

The simplest case to deal with is when the client informs the server whether the entity is new or existing

With auto-generated keys

The value of an automatically generated key can often be used to determine whether an entity needs to be inserted or updated. If the key has not been set (that is, it still has the CLR default value of null, zero, etc.), then the entity must be new and needs inserting. On the other hand, if the key value has been set, then it must have already been previously saved and now needs updating

```
public static bool IsItNew(Blog blog)
    => blog.BlogId == 0;
```

However, EF also has a built-in way to do this for any entity type and key type. Keys are set as soon as entities are tracked by the context, even if the entity is in the Added state

```
public static bool IsItNew(DbContext context, object entity)
    => !context.Entry(entity).IsKeySet;
```

With other keys

Query for the entity

```
public static bool IsItNew(BloggingContext context, Blog blog)
    => context.Blogs.Find(blog.BlogId) == null;
```

Saving single entities

If it is known whether or not an insert or update is needed, then either Add or Update can be used appropriately:

```
public static void Insert(DbContext context, object entity)
{
    context.Add(entity);
    context.SaveChanges();
}

public static void Update(DbContext context, object entity)
{
    context.Update(entity);
    context.SaveChanges();
}
```

However, if the entity uses auto-generated key values, then the Update method can be used for both cases:

```
public static void InsertOrUpdate(DbContext context, object entity)
{
    context.Update(entity);
    context
```

Working with graphs

All new/all existing entities

```
var blog = new Blog
{
    Url = "http://sample.com", Posts = new List<Post> {
       new Post { Title = "Post 1" }, new Post { Title = "Post 2" }, }
};

public static void InsertGraph(DbContext context, object rootEntity)
{
    context.Add(rootEntity);
    context.SaveChanges();
}
```

The call to Add will mark the blog and all the posts to be inserted
Likewise, if all the entities in a graph need to be updated, then Update can be used:

```
public static void UpdateGraph(DbContext context, object rootEntity)
{
    context.Update(rootEntity);
```

```
    context.SaveChanges();
}
```

Mix of new and existing entities

Update can again be used for both inserts and updates

```
public static void InsertOrUpdateGraph(DbContext context, object rootEntity)
{
    context.Update(rootEntity);
    context.SaveChanges();
}
```

Testing code that uses EF Core

The only way to be sure you are testing what runs in production is to use the same database system. For example, if the deployed application uses SQL Azure, then testing should also be done against SQL Azure

LocalDB

LocalDB is not without its issues:

- ➔ It doesn't support everything that SQL Server Developer Edition does.
- ➔ It isn't available on Linux.
- ➔ It can cause lag on the first test run as the service is spun up.

Running SQL Server (or any other database system) in a Docker container (or similar) is another way to avoid running the database system directly on your development machine

SQLite

SQLite is a good choice because:

- ➔ It runs in-process with your application and so has low overhead.
- ➔ It uses simple, automatically created files for databases, and so doesn't require database management.
- ➔ It has an in-memory mode that avoids even the file creation.

However, remember that:

- ➔ SQLite inevitably doesn't support everything that your production database system does.
- ➔ SQLite will behave differently than your production database system for some queries.
- ➔ So if you do use SQLite for some testing, make sure to also test against your real database system.

The EF Core in-memory database

This database is in general not suitable for testing applications that use EF Core. Specifically:

- ➔ It is not a relational database.
- ➔ It doesn't support transactions.

- ➔ It cannot run raw SQL queries.
- ➔ It is not optimized for performance.

None of this is very important when testing *EF Core internals* because we use it specifically where the database is irrelevant to the test

Unit testing

We use test doubles for internal testing of EF Core. However, we never try to mock DbContext or IQueryable. Doing so is difficult, cumbersome, and fragile. **Don't do it**.

Sample Code

```
[Fact]
public void Can_get_items()
{
    using (var context = new ItemsContext(ContextOptions))
    {
        var controller = new ItemsController(context);

        var items = controller.Get().ToList();

        Assert.Equal(3, items.Count);
        Assert.Equal("ItemOne", items[0].Name);
        Assert.Equal("ItemThree", items[1].Name);
        Assert.Equal("ItemTwo", items[2].Name);
    }
}
```

Performance

Efficient Querying

Project only properties you need

```
foreach (var blog in context.Blogs)
{
    Console.WriteLine("Blog: " + blog.Url);
}
```

Although this code only actually needs each Blog's Url property, the entire Blog entity is fetched, and unneeded columns are transferred from the database:

```
SELECT [b].[BlogId], [b].[CreationDate], [b].[Name], [b].[Rating], [b].[Url]
FROM [Blogs] AS [b]
```

This can be optimized by using Select to tell EF which columns to project out:

```
foreach (var blogName in context.Blogs.Select(b => b.Url))
{
    Console.WriteLine("Blog: " + blogName);
}
```

The resulting SQL pulls back only the needed columns:

```
SELECT [b].[Url] FROM [Blogs] AS [b]
```

➔ If you need to project out more than one column, project out to a C# anonymous type with the properties you want

Limit the resultset size

By default, a query returns all rows that matches its filters:

```
var blogsAll = context.Posts
    .Where(p => p.Title.StartsWith("A"))
    .ToList();
```

It's usually worth giving thought to limiting the number of results:

```
var blogs25 = context.Posts
    .Where(p => p.Title.StartsWith("A"))
    .Take(25)
    .ToList();
```

At a minimum, your UI could show a message indicating that more rows may exist in the database (and allow retrieving them in some other manner). A full-blown solution would implement paging, where your UI only shows a certain number of rows at a time, and allow users to advance to the next page as needed; this typically combines the Take and Skip operators to select a specific range in the resultset each time

Avoid cartesian explosion when loading related entities

In relational databases, all related entities are loaded by introducing JOINs in a single query. If a typical blog has multiple related posts, rows for these posts will duplicate the blog's information. This duplication leads to the so-called "cartesian explosion" problem

```
SELECT [b].[BlogId], [b].[OwnerId], [b].[Rating], [b].[Url], [p].[PostId],
[p].[AuthorId], [p].[BlogId], [p].[Content], [p].[Rating], [p].[Title]
FROM [Blogs] AS [b]
LEFT JOIN [Post] AS [p] ON [b].[BlogId] = [p].[BlogId]
ORDER BY [b].[BlogId], [p].[PostId]
```

EF allows avoiding this effect via the use of "split queries", which load the related entities via separate queries

Load related entities eagerly when possible

It is always better to use eager loading, so that EF can fetch all the required data in one roundtrip.

Beware of lazy loading

Lazy loading avoids loading related entities that aren't needed (like explicit loading), and seemingly frees the programmer from having to deal with related entities altogether. However, lazy loading is particularly prone for producing unneeded extra roundtrips which can slow the application

Buffering and streaming

```
// ToList and ToArray cause the entire resultset to be buffered:
var blogsList = context.Posts.Where(p => p.Title.StartsWith("A")).ToList();
var blogsArray = context.Posts.Where(p => p.Title.StartsWith("A")).ToArray();

// Foreach streams, processing one row at a time:
foreach (var blog in context.Posts.Where(p => p.Title.StartsWith("A")))
{
    ...
}
```

If your queries return just a few results, then you probably don't have to worry about this. However, if your query might return large numbers of rows, it's worth giving thought to streaming instead of buffering

Tracking, no-tracking and identity resolution

EF internally maintains a dictionary of tracked instances. Before handing a loaded instance to the application, EF snapshots that instance and keeps the snapshot internally. When SaveChanges is called, the application's instance is compared with the snapshot to discover the changes to be persisted. The snapshot takes up more memory, and the snapshotting process itself takes time. In read-only scenarios where changes aren't saved back to the database, the above overheads can be avoided by using no-tracking queries

Using raw SQL

- ➔ In some cases, more optimized SQL exists for your query, which EF does not generate
- ➔ Use raw SQL directly in your query, e.g. via FromSqlRaw. EF even lets you compose over the raw SQL with regular LINQ queries, allowing you to express only a part of the query in raw SQL.
- ➔ Define a user-defined function (UDF), and then call that from your queries
- ➔ Define a database view and query from it in your queries

Asynchronous programming

As a general rule, in order for your application to be scalable, it's important to always use asynchronous APIs rather than synchronous one (e.g. SaveChangesAsync rather than SaveChanges). Synchronous APIs block the thread for the duration of database I/O, increasing the need for threads and the number of thread context switches that must occur

Warning: Avoid mixing synchronous and asynchronous code in the same application - it's very easy to inadvertently trigger subtle thread-pool starvation issues

Efficient Updating

```
foreach (var employee in context.Employees)
{
    employee.Salary += 1000;
}
```

While this is perfectly valid code, let's analyze what it does from a performance perspective:

- ➔ A database round trip is performed, to load all the relevant employees

- ➔ EF Core's change tracking creates snapshots when loading the entities
- ➔ A second database roundtrip is performed to save all the changes. While all changes are done in a single roundtrip thanks to batching, EF Core still sends an UPDATE statement per employee, which must be executed by the database

Unfortunately, EF doesn't currently provide APIs for performing bulk updates. Until these are introduced, you can use raw SQL to perform the operation where performance is sensitive

```
context.Database.ExecuteSqlRaw("UPDATE [Employees] SET [Salary] = [Salary] +
1000");
```

Modeling for Performance

Denormalization and caching

- ➔ Denormalization is the practice of adding redundant data to your schema, usually in order to eliminate joins when querying
- ➔ For example, for a model with Blogs and Posts, where each Post has a Rating, you may be required to frequently show the average rating of the Blog.
- ➔ Denormalization would add the calculated average of all posts to a new column on Blog, so that it is immediately accessible, without joining or calculating
- ➔ The above can be viewed as a form of caching - aggregate information from the Posts is cached on their Blog

Stored computed columns

For example, a Customer may have FirstName and LastName columns, but we may need to search by the customer's full name. A stored computed column is automatically maintained by the database - which recalculates it whenever the row is changed - and you can even define an index over it to speed up queries.

Materialized views

Materialized views are similar to regular views, except that their data is stored on disk ("materialized"), rather than calculated every time when the view is queried

Advanced Performance Topics

DbContext pooling

Context pooling can increase throughput in high-scale scenarios such as web servers by reusing context instances, rather than creating new instances for each request

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

When AddDbContextPool is used, at the time a context instance is requested, EF first checks 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

Context pooling works by reusing the same context instance across requests. This means that it's effectively registered as a Singleton in terms of the instance itself so that it's able to persist

Warning: Avoid using context pooling in apps that maintain state. For example, private fields in the context that shouldn't be shared across requests

Asynchronous Programming

Asynchronous operations avoid blocking a thread while the query is executed in the database. Async operations are important for keeping a responsive UI in rich client applications. EF Core doesn't support multiple parallel operations being run on the same context instance. You should always wait for an operation to complete before beginning the next operation

Async LINQ operators

Asynchronous LINQ operators include ToListAsync, SingleAsync, AsAsyncEnumerable

```
var blogs = await context.Blogs.Where(b => b.Rating > 3).ToListAsync();
```

Note that there are no async versions of some LINQ operators such as Where or OrderBy, because these only build up the LINQ expression tree and don't cause the query to be executed in the database

Client-side async LINQ operators

The async LINQ operators discussed above can only be used on EF queries - you cannot use them with client-side LINQ to Objects query. To perform client-side async LINQ operations outside of EF, use the `System.Linq.Async` package

```
var groupedHighlyRatedBlogs = await context.Blogs
    .AsQueryable()
    .Where(b => b.Rating > 3) // server-evaluated
    .AsAsyncEnumerable()
    .GroupBy(b => b.Rating) // client-evaluated
    .ToListAsync();
```

Connection Resiliency

Connection resiliency automatically retries failed database commands. Note that enabling retry on failure causes EF to internally buffer the resultset, which may significantly increase memory requirements for queries returning large result sets.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .UseSqlServer(
             @"Server=(localdb)\mssqllocaldb;Database=EFMiscellanous
             .ConnectionResiliency;Trusted_Connection=True;ConnectRetryCount=0",
             options => options.EnableRetryOnFailure());
}
```

or in Startup.cs for an ASP.NET Core application

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<PicnicContext>(
        options => options.UseSqlServer(
            "<connection string>",
            providerOptions => providerOptions.EnableRetryOnFailure()));
}
```

Custom execution strategy

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .UseMyProvider(
            "<connection string>",
            options => options.ExecutionStrategy(...));
}
```

Execution strategies and transactions

if your code initiates a transaction using BeginTransaction() you are defining your own group of operations that need to be treated as a unit, and everything inside the transaction would need to be played back shall a failure occur. You will receive an exception

The solution is to manually invoke the execution strategy with a delegate representing everything that needs to be executed. If a transient failure occurs, the execution strategy will invoke the delegate again.

```
using (var db = new BloggingContext())
{
    var strategy = db.Database.CreateExecutionStrategy();

    strategy.Execute(
        () =>
        {
            using (var context = new BloggingContext())
            {
                using (var transaction = context.Database.BeginTransaction())
                {
                    context.Blogs.Add(new Blog { Url =
"http://blogs.msdn.com/dotnet" });
                    context.SaveChanges();

                    context.Blogs.Add(new Blog { Url =
"http://blogs.msdn.com/visualstudio" });
                    context.SaveChanges();

                    transaction.Commit();
                }
            }
        });
}
```

Transaction commit failure and the idempotency issue

In general, when there is a connection failure the current transaction is rolled back. However, if the connection is dropped while the transaction is being committed the resulting state of the transaction is unknown

By default, the execution strategy will retry the operation as if the transaction was rolled back, but if it's not the case this will result in an exception if the new database state is incompatible or could lead to **data corruption** if the operation does not rely on a particular state, for example when inserting a new row with auto-generated key values

More:
https://docs.microsoft.com/en-us/ef/core/miscellaneous/connection-resiliency#transaction-commit-failure-and-the-idempotency-issue

Connection Strings

In ASP.NET Core the configuration system is very flexible, and the connection string could be stored in appsettings.json, an environment variable, the user secret store, or another configuration source

You can use the Secret Manager tool to store your database password and then, in scaffolding, use a connection string that simply consists of Name=<database-alias>.

```
.dotnet user-secrets set ConnectionStrings:YourDatabaseAlias "Data
Source=(localdb)\MSSQLLocalDB;Initial Catalog=YourDatabase"
dotnet ef dbcontext scaffold Name=ConnectionStrings:YourDatabaseAlias
Microsoft.EntityFrameworkCore.SqlServer
```

EoF