C# Entity Framework Core

Notes by rajeeshvengalapurath@gmail.com
8 Jan 20

- EF Core can serve as an **object-relational mapper (O/RM)**, enabling .NET developers to **work with a database using .NET objects**, and eliminating the need for most of the data-access code they usually need to write.
    - **An object-relational mapper (ORM)** is a code library that automates the **transfer of data** stored in **relational databases tables** into **objects** that are more commonly used in application code

Links

- FK to the Same Table Code First Entity Framework https://stackoverflow.com/questions/29516342/fk-to-the-same-table-code-first-entity-framework

Supported database engines

https://docs.microsoft.com/en-us/ef/core/providers/index?tabs=dotnet-core-cli

- SQL Server 2012 onwards
- SQLite 3.7 onwards
- EF Core in-memory database
- Azure Cosmos DB SQL API
- PostgreSQL
- MySQL, MariaDB
- MySQL 5 onwards
- Oracle DB 9.2.0.4 onwards
- PostgreSQL 8.0 onwards
- SQLite 3 onwards
- Stores data in files
- Microsoft Access files
- SQL Server Compact 3.5
- SQL Server Compact 4.0
- Firebird 2.5 and 3.x
- Teradata Database 16.10 onwards
- Firebird 2.5 and 3.x
- Progress OpenEdge
- MySQL
- Oracle DB 11.2 onwards
- Db2, Informix
- MyCAT Server

Adding a database provider to your application

- **dotnet add** package provider_package_name
- Once installed, you will configure the provider in your **DbContext**, either in the **OnConfiguring** method or in the **AddDbContext** method if you are using a dependency injection container
- Example, configures the SQL Server provider with the passed connection string
    - `optionsBuilder.UseSqlServer("Server=(localdb)\mssqllocaldb;Database=My Database;Trusted_Connection=True;");`

Getting Started with EF Core

Prerequisites

.NET Core 3.0 SDK. (https://www.microsoft.com/net/download/core)

Create a new project

```
dotnet new console -o EFGetStarted
cd EFGetStarted
```

Install Entity Framework Core

PowerShell

```
Install-Package Microsoft.EntityFrameworkCore.Sqlite
```

Create the model

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

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

        protected override void OnConfiguring(DbContextOptionsBuilder options)
            => options.UseSqlite("Data Source=blogging.db");
    }

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

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

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

Create the database

The following steps use migrations to create a database

```
Install-Package Microsoft.EntityFrameworkCore.Tools
Add-Migration InitialCreate
Update-Database
```

Create, read, update & delete

```
using System;
using System.Linq;

namespace EFGetStarted
{
    class Program
    {
        static void Main()
        {
            using (var db = new BloggingContext())
            {
                // Create
                Console.WriteLine("Inserting a new blog");
                db.Add(new Blog { Url = "http://blogs.msdn.com/adonet" });
                db.SaveChanges();

                // Read
                Console.WriteLine("Querying for a blog");
                var blog = db.Blogs
                    .OrderBy(b => b.BlogId)
                    .First();

                // Update
                Console.WriteLine("Updating the blog and adding a post");
                blog.Url = "https://devblogs.microsoft.com/dotnet";
                blog.Posts.Add(
                    new Post
                    {
                        Title = "Hello World",
```

```
                    Content = "I wrote an app using EF Core!"
                });
            db.SaveChanges();

            // Delete
            Console.WriteLine("Delete the blog");
            db.Remove(blog);
            db.SaveChanges();
        }
    }
  }
}
```

Connection Strings

- If your connection string contains sensitive information, such as username and password, you can protect the contents of the configuration file using the Secret Manager tool(https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets?view=aspnetcore-3.1&tabs=windows#secret-manager)
- The **providerName** setting is **not required on EF Core** connection strings stored in App.config because the database provider is configured via code
- You can then **read** the connection string using the **ConfigurationManager** API in your **context's OnConfiguring** method
- In ASP.NET Core the connection string could be **stored** in
    - Appsettings.json
    - an environment variable
    - the user secret store
    - another configuration source

Code

DbContext

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

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
            optionsBuilder.UseSqlServer(
       ConfigurationManager.ConnectionStrings["BloggingDatabase"].ConnectionString)
       ;
    }
}
```

Startup.cs

The context is typically configured in Startup.cs

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<BloggingContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("BloggingDatabase")));
}
```

Connection Resiliency

- Connection resiliency **automatically retries failed database commands**

Transaction commit failure and the idempotency issue

- 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

There are several ways to deal with this

- Rebuild application state
    - Discard the current DbContext.
    - Create a new DbContext and restore the state of your application from the database.
    - Inform the user that the last operation might not have been completed successfully.
- Add state verification
    - ```
      using (var db = new BloggingContext())
      {
          var strategy = db.Database.CreateExecutionStrategy();

          var blogToAdd = new Blog {Url = "http://blogs.msdn.com/dotnet"};
          db.Blogs.Add(blogToAdd);

          strategy.ExecuteInTransaction(db,
              operation: context =>
              {
                  context.SaveChanges(acceptAllChangesOnSuccess: false);
              },
              verifySucceeded: context => context.Blogs.AsNoTracking().Any(b
      => b.BlogId == blogToAdd.BlogId));
      ```
      ```

```
            db.ChangeTracker.AcceptAllChanges();
        }
```

- Manually track the transaction

  If you need to use store-generated keys or need a generic way of handling commit failures that doesn't depend on the operation performed each transaction could be assigned an ID that is checked when the commit fails.

  - Add a table to the database used to track the status of the transactions.
  - Insert a row into the table at the beginning of each transaction.
  - If the connection fails during the commit, check for the presence of the corresponding row in the database.
  - If the commit is successful, delete the corresponding row to avoid the growth of the table.

DbContext

Configuring

- DbContext **must have** an instance of **DbContextOptions** in order to perform any work
- The DbContextOptions can be supplied to the DbContext by overriding the **OnConfiguring method** or externally via a **constructor argument**.
- If both are used, **OnConfiguring is applied last** and can overwrite options supplied to the constructor argument

Constructor argument

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

    public DbSet<Blog> Blogs { get; set; }
}
```

Your application can now pass the DbContextOptions when instantiating a context, as follows

```
var optionsBuilder = new DbContextOptionsBuilder<BloggingContext>();
optionsBuilder.UseSqlite("Data Source=blog.db");

using (var context = new BloggingContext(optionsBuilder.Options))
{
  // do stuff
}
```

OnConfiguring

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

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlite("Data Source=blog.db");
    }
}
```

An application can simply instantiate such a context without passing anything to its constructor

```
using (var context = new BloggingContext())
{
  // do stuff
}
```

Using DbContext with dependency injection

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<BloggingContext>(options => options.UseSqlite("Data
Source=blog.db"));
}
```

This requires adding a constructor argument to your DbContext type that accepts DbContextOptions<TContext>.
Context code:

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

    public DbSet<Blog> Blogs { get; set; }
}
```

Application code (in ASP.NET Core):

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

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

```
    ...
}
```

Avoiding DbContext threading issues

- Entity Framework Core **does not support multiple parallel operations** being run on the same DbContext instance. This includes both parallel execution of async queries and any explicit concurrent use from multiple threads.
    - 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** with a message
- When concurrent access **goes undetected**, it can result in undefined behavior, **application crashes** and **data corruption**
- **Asynchronous methods** enable EF Core to initiate operations that access the database **in a non-blocking way**
- But **if 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**
- **Always await EF Core asynchronous methods immediately**
- The **AddDbContext** extension method registers DbContext types with a **scoped lifetime** by default
    - This is safe from concurrent access issues in ASP.NET Core applications because there is only one thread executing each client request at a given time, and because each request gets a separate dependency injection scope (and therefore a separate DbContext instance)
- However any code that explicitly executes multiple threads in parallel should ensure that DbContext instances aren't ever accessed concurrently
    - Using dependency injection, this can be achieved by either registering the context as scoped and creating scopes (using IServiceScopeFactory) for each thread, or by registering the DbContext as transient (using the overload of AddDbContext which takes a ServiceLifetime parameter)

Model

Creating and configuring a model

Covers configuration that can be applied to a model targeting any data store and that which can be applied when targeting any relational database

Use fluent API to configure a model

- This is the **most powerful method** of configuration and allows configuration to be specified without modifying your entity classes.

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

```
using Microsoft.EntityFrameworkCore;

namespace EFModeling.FluentAPI.Required
{
    class MyContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }

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

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

Use data annotations to configure a model

- Data annotations will **override conventions**, but **will be overridden by Fluent API** configuration

```
using Microsoft.EntityFrameworkCore;
using System.ComponentModel.DataAnnotations;

namespace EFModeling.DataAnnotations.Required
{
    class MyContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
    }

    #region Required
    public class Blog
    {
        public int BlogId { get; set; }
        [Required]
        public string Url { get; set; }
    }
    #endregion
}
```

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.

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

    public Blog Blog { get; set; }
}

public class AuditEntry
{
    public int AuditEntryId { get; set; }
    public string Username { get; set; }
    public string Action { get; set; }
}
```

Excluding types from the model

```
[NotMapped]
public class BlogMetadata
{
    public DateTime LoadedFromDatabase { get; set; }
}
```

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

Table name

```
[Table("blogs")]
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
//Fluent API
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .ToTable("blogs");
}
```

Table schema

```
[Table("blogs", Schema = "blogging")]
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

//Fluent

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

Rather than specifying the schema for each table, you can also define the default schema at the model level with the fluent API:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.HasDefaultSchema("blogging");
}
```

Entity Properties

Included and excluded properties

By convention, all public properties with a getter and a setter will be included in the model.

Specific properties can be excluded as follows:

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

    [NotMapped]
    public DateTime LoadedFromDatabase { get; set; }
}
//or
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Ignore(b => b.LoadedFromDatabase);
}
```

Column names

```
public class Blog
{
    [Column("blog_id")]
    public int BlogId { get; set; }
    public string Url { get; set; }
}
//or
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.BlogId)
        .HasColumnName("blog_id");
}
```

Column data types

```
public class Blog
{
    public int BlogId { get; set; }
    [Column(TypeName = "varchar(200)")]
    public string Url { get; set; }
    [Column(TypeName = "decimal(5, 2)")]
    public decimal Rating { get; set; }
}
//or
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

```
public class Blog
{
    public int BlogId { get; set; }
    [MaxLength(500)]
    public string Url { get; set; }
}
//or
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .HasMaxLength(500);
}
```

Required and optional properties

When mapping to a relational database schema, required properties are created as non-nullable columns, and optional properties are created as nullable columns

```
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
}
//or
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

    public Customer(string firstName, string lastName, string? middleName = null)
    {
        FirstName = firstName;
        LastName = lastName;
        MiddleName = middleName;
    }
}
```

Explicit configuration

```
public class Blog
{
```

```
    public int BlogId { get; set; }
    [Required]
    public string Url { get; set; }
}
//or
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .IsRequired();
}
```

Keys

```
class Car
{
    [Key]
    public string LicensePlate { get; set; }

    public string Make { get; set; }
    public string Model { get; set; }
}
//or
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Car>()
        .HasKey(c => c.LicensePlate);
}
```

Composite keys

Composite keys can only be configured using the **Fluent API**; conventions will never setup a composite key, and you can not use Data Annotations to configure one

```
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

No value generation

No value generation means that **you will always supply** a valid value to be saved to the database. This valid value must be assigned to new entities before they are added to the context

Value generated on add

Depending on the database provider being used, values may be generated client side by EF or in the database.

Value generated on add or update

Value generated on add or update means that a new value is generated every time the record is saved (insert or update). How the value is generated for added and updated entities will depend on the database provider being used

Value generated on add

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public DateTime Inserted { get; set; }
}
```

Default values

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Rating)
        .HasDefaultValue(3);
}
```

You can also specify a SQL fragment that is used to calculate the default value

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Created)
        .HasDefaultValueSql("getdate()");
}
```

Value generated on add or update

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
    public DateTime LastUpdated { get; set; }
}
//or
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.LastUpdated)
        .ValueGeneratedOnAddOrUpdate();
}
```

Computed columns

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Person>()
        .Property(p => p.DisplayName)
        .HasComputedColumnSql("[LastName] + ', ' + [FirstName]");
}
```

No value generation

Disabling value generation on a property is typically necessary if a convention configures it for value generation. For example, if you have a primary key of type int, it will be implicitly set configured as value generated on add; you can disable this via the following

```
public class Blog
{
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public int BlogId { get; set; }
    public string Url { get; set; }
}
//or
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.BlogId)
        .ValueGeneratedNever();
}
```

Timestamp/row version

A timestamp is a property where a new value is generated by the database every time a row is inserted or updated. The property is also treated as a **concurrency token.** This ensures you will **get an exception** if anyone else has modified a row that you are trying to update since you queried for the data

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

    public string Url { get; set; }

    [Timestamp]
    public byte[] Timestamp { get; set; }
}
//or
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(p => p.Timestamp)
            .IsRowVersion();
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    public byte[] Timestamp { get; set; }
}
```

Shadow Properties

Shadow properties are properties that are not defined in your .NET entity class but are defined for that entity type in the EF Core model. The value and state of these properties is maintained purely in the Change Tracker

Shadow properties are useful when there is data in the database that should not be exposed on the mapped entity types. They are most often used for foreign key properties, where the relationship between two entities is represented by a **foreign key value in the database**, but the relationship is managed on the entity types using navigation properties between the entity types.

For example, the following code listing will result in a **BlogId** shadow property being introduced to the Post entity

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

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

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

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

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

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.
- **Foreign key**: The properties in the dependent entity that are used to store the principal key values for the related entity.
- **Principal key**: The properties that uniquely identify the principal entity. This may be the primary key or an alternate key.
- **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.

The following code shows a one-to-many relationship between Blog and Post

```
public class Blog //Blog is the principal entity
{
    public int BlogId { get; set; } //Blog.BlogId is the principal key
    public string Url { get; set; }

    public List<Post> Posts { get; set; } //Blog.Posts is a collection navigation
                                          property
}
```

```
public class Post //Post is the dependent entity
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; } //Post.BlogId is the foreign key
    public Blog Blog { get; set; }  //Post.Blog is a reference navigation property
                                    //Post.Blog is the inverse navigation property
                                         of Blog.Posts (and vice versa)
}
```

Conventions

Fully defined relationships

The most common pattern for relationships is to have navigation properties defined on both ends of the relationship and a foreign key property defined in the dependent entity class

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

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

No foreign key property

While it is recommended to have a foreign key property defined in the dependent entity class, it is not required

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

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

public class Post
{
```

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

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

Single navigation property

Including just one navigation property (no inverse navigation, and no foreign key property) is enough to have a relationship defined by convention

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

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

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

Cascade delete

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

Manual configuration

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Post>()
            .HasOne(p => p.Blog)
            .WithMany(b => b.Posts);
    }
}

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

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

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

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

Single navigation property

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasMany(b => b.Posts)
            .WithOne();
    }
}

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

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

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

Foreign key

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Post>()
            .HasOne(p => p.Blog)
            .WithMany(b => b.Posts)
```

```
            .HasForeignKey(p => p.BlogForeignKey);
    }
}

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

    public List<Post> Posts { get; set; }
}
//or
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

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

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

    public int BlogForeignKey { get; set; }

    [ForeignKey("BlogForeignKey")]
    public Blog Blog { get; set; }
}
```

Foreign key constraint name

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Post>()
        .HasOne(p => p.Blog)
        .WithMany(b => b.Posts)
        .HasForeignKey(p => p.BlogId)
        .HasConstraintName("ForeignKey_Post_Blog");
}
```

Without navigation property

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

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

Cascade delete

You can use the Fluent API to configure the cascade delete behavior for a given relationship explicitly

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Post>()
        .HasOne(p => p.Blog)
        .WithMany(b => b.Posts)
        .OnDelete(DeleteBehavior.Cascade);
}
```

One-to-one

One to one relationships have a reference navigation property on both sides. They follow the same conventions as one-to-many relationships, but a unique index is introduced on the foreign key property to ensure only one dependent is related to each principal

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasOne(b => b.BlogImage)
            .WithOne(i => i.Blog)
            .HasForeignKey<BlogImage>(b => b.BlogForeignKey);
    }
}
```

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

    public BlogImage BlogImage { get; set; }
}

public class BlogImage
{
    public int BlogImageId { get; set; }
    public byte[] Image { get; set; }
    public string Caption { get; set; }

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

Many-to-many

```
class MyContext : DbContext
{
    public DbSet<Post> Posts { get; set; }
    public DbSet<Tag> Tags { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<PostTag>()
            .HasKey(t => new { t.PostId, t.TagId });

        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);
    }
}

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

    public List<PostTag> PostTags { get; set; }
}

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

```
    public List<PostTag> PostTags { get; set; }
}

public class PostTag
{
    public int PostId { get; set; }
    public Post Post { get; set; }

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

Indexes

- Indexes **cannot** be created using **data annotations**
- By default, indexes **aren't unique**
- By convention, an **index is created** in each property (or set of properties) that are used as a **foreign key**

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .HasIndex(b => b.Url);
}
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Person>()
        .HasIndex(p => new { p.FirstName, p.LastName });
}
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .HasIndex(b => b.Url)
        .IsUnique();
}
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .HasIndex(b => b.Url)
        .HasName("Index_Url");
}
```

Sequences

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.HasSequence<int>("OrderNumbers");

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

```
}
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.HasSequence<int>("OrderNumbers", schema: "shared")
        .StartsAt(1000)
        .IncrementsBy(5);
}
```

Backing Fields

Conventions

```
public class Blog
{
    private string _url;

    public int BlogId { get; set; }

    public string Url
    {
        get { return _url; }
        set { _url = value; }
    }
}
```

Configure a backing field for a property

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

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

public class Blog
{
    private string _validatedUrl;

    public int BlogId { get; set; }

    public string Url
    {
        get { return _validatedUrl; }
    }

    public void SetUrl(string url)
    {
        using (var client = new HttpClient())
```

```
        {
            var response = client.GetAsync(url).Result;
            response.EnsureSuccessStatusCode();
        }

        _validatedUrl = url;
    }
}
```

Value Conversions

- From one value to another of the **same type**
    - For example, encrypting strings
- From a value of one type to a value of **another type**
    - For example, converting enum values to and from strings in the database
    -

Value converters are specified in terms of a **ModelClrType** and a **ProviderClrType**. The model type is the .NET type of the property in the entity type. The provider type is the .NET type understood by the database provider

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

public enum EquineBeast
{
    Donkey,
    Mule,
    Horse,
    Unicorn
}

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder
        .Entity<Rider>()
        .Property(e => e.Mount)
        .HasConversion(
            v => v.ToString(), //convertToProviderExpression
            v => (EquineBeast)Enum.Parse(typeof(EquineBeast), v));
                                //convertFromoProviderExpression
}
```

The ValueConverter class

```
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 converter classes

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

- Notice that EnumToStringConverter is included in this list. This means that there is no need to specify the conversion explicitly, as shown above. Instead, just use the built-in converter

```
var converter = new EnumToStringConverter<EquineBeast>();

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

Pre-defined conversions

```
modelBuilder
    .Entity<Rider>()
    .Property(e => e.Mount)
    .HasConversion<string>();
//or
public class Rider
```

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

    [Column(TypeName = "nvarchar(24)")]
    public EquineBeast Mount { get; set; }
}
```

Limitations

- Null cannot be converted.
- There is currently no way to spread a conversion of one property to multiple columns or vice-versa.
- Use of value conversions may impact the ability of EF Core to translate expressions to SQL. A warning will be logged for such cases. Removal of these limitations is being considered for a future release.

Data Seeding

Data seeding is the process of populating a database with an **initial set of data**. There are several ways this can be accomplished in EF Core:

- Model seed data
- Manual migration customization
- Custom initialization logic

Unlike in **EF6**, in **EF Core**, seeding data can be associated with an entity type as part of the model configuration

As an example, this will configure seed data for a Blog in OnModelCreating:

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

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"
});
```

If the entity type has any properties in shadow state an anonymous class can be used to provide the values:

```
modelBuilder.Entity<Post>().HasData(
    new { BlogId = 1, PostId = 2, Title = "Second post", Content = "Test 2" });
```

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" });
```

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

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();
}
```

Depending on the constraints of your deployment the initialization code can be executed in different ways:

- Running the initialization app locally
- Deploying the initialization app with the main app, invoking the initialization routine and disabling or removing the initialization app.

Keyless Entity Types (Views etc)

Keyless entity types can be used to carry out database queries against data that doesn't contain key values

Difference with regular entity types

- Cannot have a key defined.
- Are never tracked for changes in the DbContext and therefore are never inserted, updated or deleted on the database.
- Are never discovered by convention.
- Only support a subset of navigation mapping capabilities, specifically:
    - They may never act as the principal end of a relationship.
    - They may not have navigations to owned entities
    - They can only contain reference navigation properties pointing to regular entities.
    - Entities cannot contain navigation properties to keyless entity types.

- Need to be configured with .HasNoKey() method call.
- May be mapped to a defining query. A defining query is a query declared in the model that acts as a data source for a keyless entity type.

Usage scenarios

- Serving as the return type for raw SQL queries.
- Mapping to database views that do not contain a primary key.
- Mapping to tables that do not have a primary key defined.
- Mapping to queries defined in the model.

Mapping to database objects

Mapping a keyless entity type to a database object is achieved using the ToTable or ToView fluent API. Database object is actually not required to be a database view. It can alternatively be a database table that will be treated as read-only

```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }
    public ICollection<Post> Posts { 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; }
}

db.Database.ExecuteSqlRaw(
    @"CREATE VIEW View_BlogPostCounts AS
        SELECT b.Name, Count(p.PostId) as PostCount
        FROM Blogs b
        JOIN Posts p on p.BlogId = b.BlogId
        GROUP BY b.Name");

public class BlogPostsCount
{
    public string BlogName { get; set; }
    public int PostCount { get; set; }
}

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

```
        .Entity<BlogPostsCount>(eb =>
        {
            eb.HasNoKey();
            eb.ToView("View_BlogPostCounts");
            eb.Property(v => v.BlogName).HasColumnName("Name");
        });
}
```

Next, we configure the DbContext to include the DbSet<T>:

```
public DbSet<BlogPostsCount> BlogPostCounts { get; set; }
```

Finally, we can query the database view in the standard way:

```
var postCounts = db.BlogPostCounts.ToList();

foreach (var postCount in postCounts)
{
    Console.WriteLine($"{postCount.BlogName} has {postCount.PostCount} posts.");
    Console.WriteLine();
}
```

Concurrency Tokens

```
public class Person
{
    public int PersonId { get; set; }

    [ConcurrencyCheck]
    public string LastName { get; set; }

    public string FirstName { get; set; }
}
```

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. The precise details depend on the database provider being used; for SQL Server, a byte[] property is usually used, which will be set up as a ROWVERSION column in the database.

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

    public string Url { get; set; }

    [Timestamp]
    public byte[] Timestamp { get; set; }
```

```
}
```

Managing Database Schemas

- EF Core provides two primary ways of keeping your EF Core model and database schema in sync. To choose between the two, decide
    - whether your **EF Core model** is the source of truth
    - or the **database schema** is the source of truth
- If you want your **EF Core model** to be the source of truth, use **Migrations**. As you make changes to your EF Core model, this approach incrementally applies the corresponding schema changes to your database so that it remains compatible with your EF Core model
- **Use Reverse Engineering** if you want your database schema to be the source of truth. This approach allows you to **scaffold a DbContext** and the entity type classes by reverse engineering your database schema into an EF Core model

Migrations

Create a migration

```
Add-Migration InitialCreate
```

Update the database

```
Update-Database
```

Customize migration code

Change the default migration for merging two columns by dropping columns and adding new column as shown her

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

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

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

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

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

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

Empty migrations

Sometimes it's useful to add a migration without making any model changes. In this case, adding a new migration creates code files with empty classes. You can customize this migration to perform operations that don't directly relate to the EF Core model. Some things you might want to manage this way are:

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

Remove a migration

To remove the last migration
`Remove-Migration`

Revert a migration

`Update-Database NameOfLastGoodMigrationHere`

Generate SQL scripts

`Script-Migration`
There are several options to this command.

- The **from** migration should be the last migration applied to the database before running the script. If no migrations have been applied, specify 0 (this is the default).
- The **to** migration is the last migration that will be applied to the database after running the script. This defaults to the last migration in your project.
- An **idempotent** script can optionally be generated. This script only applies migrations if they haven't already been applied to the database. This is useful if you don't exactly

know what the last migration applied to the database was or if you are deploying to multiple databases that may each be at a different migration.

Apply migrations at runtime

```
myDbContext.Database.Migrate();
```

Warning

- This approach isn't for everyone. While it's great for apps with a local database, most applications will require more **robust deployment strategy like generating SQL scripts**.
- Don't call EnsureCreated() before Migrate(). EnsureCreated() bypasses Migrations to create the schema, which causes Migrate() to fail.

Using a Separate Migrations Project

1. Create a new class library.
2. Add a reference to your DbContext assembly.
3. Move the migrations and model snapshot files to the class library.
    a. Tip: If you have no existing migrations, generate one in the project containing the DbContext then move it. This is important because if the migrations assembly does not contain an existing migration, the Add-Migration command will be unable to find the DbContext.
4. Configure the migrations assembly

```
options.UseSqlServer(connectionString, x =>
x.MigrationsAssembly("MyApp.Migrations"));
```

5. Add a reference to your migrations assembly from the startup assembly.
    a. If this causes a circular dependency, update the output path of the class library:

```
<PropertyGroup>
      <OutputPath>..\MyStartupProject\bin\$(Configuration)\</OutputPath>
</PropertyGroup>
```

If you did everything correctly, you should be able to add new migrations to the project

```
dotnet ef migrations add NewMigration --project MyApp.Migrations
```

Reverse Engineering (Scaffolding)

Reverse engineering is the process of scaffolding entity type classes and a DbContext class based on a database schema. Before reverse engineering, you'll need to install either the PMC tools (Visual Studio only) \

Connection string

```
Scaffold-DbContext 'Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Chinook'
Microsoft.EntityFrameworkCore.SqlServer
```

Specifying tables

```
Scaffold-DbContext ... -Tables Artist, Album
```

Preserving names

Table and column names are fixed up to better match the .NET naming conventions for types and properties by default. Specifying the -UseDatabaseNames switch in PMC or the --use-database-names option in the CLI will disable this behavior preserving the original database names as much as possible

Fluent API or Data Annotations

Entity types are configured using the Fluent API by default. Specify -DataAnnotations (PMC) or --data-annotations (CLI) to instead use data annotations when possible

```
entity.Property(e => e.Title)
    .IsRequired()
    .HasMaxLength(160);
//or
[Required]
[StringLength(160)]
public string Title { get; set; }
```

Directories and namespaces

The entity classes and a DbContext class are scaffolded into the project's root directory and use the project's default namespace. You can specify the directory where classes are scaffolded using -OutputDir (PMC) or --output-dir (CLI). The namespace will be the root namespace plus the names of any subdirectories under the project's root directory.

You can also use -ContextDir (PMC) and --context-dir (CLI) to scaffold the DbContext class into a separate directory from the entity type classes.

```
Scaffold-DbContext ... -ContextDir Data -OutputDir Models
```

Querying Data

LINQ allows you to use C# to write **strongly typed queries**. EF Core passes a representation of the LINQ query to the database provider. Database providers in turn **translate it to database-specific query language** (for example, SQL for a relational database)

Loading all data

```
using (var context = new BloggingContext())
{
```

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

Loading a single entity

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);
}
```

Filtering

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Where(b => b.Url.Contains("dotnet"))
        .ToList();
}
```

Client evaluation in the top-level projection

In the following example, a helper method is used to standardize URLs for blogs, which are returned from a SQL Server database. Since the SQL Server provider has no insight into how this method is implemented, it isn't possible to translate it into SQL. All other aspects of the query are evaluated in the database, but passing the returned URL through this method is done on the client.

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

public static string StandardizeUrl(string url)
{
    url = url.ToLower();

    if (!url.StartsWith("http://"))
    {
        url = string.Concat("http://", url);
    }

    return url;
}
```

Complex Query Operators

Not all LINQ operators have suitable translations on the server side. Sometimes, a query in one form translates to the server but if written in a different form doesn't translate even if the result is the same

- the query variable itself only stores the query commands. The actual execution of the query is deferred until you iterate over the query variable in a foreach statement
- Because the query variable itself never holds the query results, you can execute it as often as you like
- In your application, you could create one query that retrieves the latest data, and you could execute it repeatedly at some interval to retrieve different results every time

Inner Join

2 tables

```
var result = Blog.Blogs().Join(Post.Posts()
       , blog => blog.BlogID
       , post => post.BlogID
       , (blog, post) => new { blog.Title, post.Content}
                           );

result = from blog in Blog.Blogs()
         join post in Post.Posts()
              on blog.BlogID equals post.BlogID
         select new { blog.Title, post.Content };
```

3 tables

```
var result = Blog.Blogs().Join(
       Category.Categories()
       , blog => blog.CategoryID
       , category => category.CategoryID
       , (blog, category) => new { blog, category }
       ).Join(Post.Posts()
              , blogCategory => blogCategory.blog.BlogID
              , post => post.BlogID
              , (blogCategory, post) => new { blogCategory.blog.Title,
                           blogCategory.category.Name, post.Content });

var result = from blog in Blog.Blogs()
             join category in Category.Categories()
             on blog.CategoryID equals category.CategoryID
             join post in Post.Posts()
             on blog.BlogID equals post.BlogID
             select new { blog.Title, category.Name, post.Content };
```

Disadvantages of method: An additional variable blogCategory is needed after first join.

Grouped Joins (into)

The group join is useful for producing hierarchical data structures. It pairs each element from the first collection with a set of correlated elements from the second collection

Each element of the first collection appears in the result set of a group join **regardless of whether correlated elements are found in the second collection**. In the case where no correlated elements are found, the sequence of correlated elements for that element is empty. The result selector therefore has access to every element of the first collection. This differs from the result selector in a non-group join, which cannot access elements from the first collection that have no match in the second collection

Result will always contain elements of first collection

```
var result = from blog in Blog.Blogs()
             join post in Post.Posts()
                   on blog.BlogID equals post.BlogID
             into posts //will contain posts associated with each blog
             select new { F1=blog.Title, F2=posts };
```

```
var result = Blog.Blogs().GroupJoin(
                Post.Posts()
                    , blog => blog.BlogID //outer key selector
                    , post => post.BlogID //inner key selector
                    , (blog, postCollection) => new { blog.Title, postCollection}
                                );
```

Above query modified below to select only “content” field of post to return as a collection of “contents” named PostContentCollection

```
var result = Blog.Blogs().GroupJoin(
             Post.Posts()
                   , blog => blog.BlogID
                   , post => post.BlogID
                   , (blog, postCollection) => new { blog.Title,
             PostContentCollection = postCollection.Select(post=>post.Content)}
                              );
```

Left Join

- You can use LINQ to perform a left outer join by calling the DefaultIfEmpty method on the results of a group join
- The first step in producing a left outer join of two collections is to perform an inner join by using a group join
- The second step is to include each element of the first (left) collection in the result set even if that element has no matches in the right collection
    - This is accomplished by calling DefaultIfEmpty on each sequence of matching elements from the group join

```
var query = products.GroupJoin(categories
       , product => product.CategoryID
       , category => category.CategoryID
       , (product, categories) => new { product, categories })
              .SelectMany(
                     groupJoinResult => groupJoinResult.categories.DefaultIfEmpty()
                     , (groupJoinResult, category) => new {
                            groupJoinResult.product.Name
                            , CategoryName=category?.Name ?? ""
                                                        });

var query = from product in products
       join category in categories
       on product.CategoryID equals category.CategoryID into categoriesGroup
       from category in categoriesGroup.DefaultIfEmpty()
       select new { product.Name, CategoryName = category?.Name ?? "" };
```

Inner Join - Left Join Method 2

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 };
```

Left Join

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

Order by

```
var result = Blog.Blogs()
             .OrderBy(blog => blog.Title)
             .Select(blog => blog.Title);

var result = Blog.Blogs()
             .OrderByDescending(blog => blog.Title)
             .Select(blog => blog.Title);

var result = from blog in Blog.Blogs()
             orderby blog.Title
             select blog.Title;

var result = from blog in Blog.Blogs()
             orderby blog.Title.Length, blog.CategoryID descending
             select blog.Title;
```

ThenBy & ThenByDescending

```
var result = Blog.Blogs()
```

```
            .OrderByDescending(blog => blog.Title)
            .ThenBy(blog=>blog.CategoryID) //ThenByDescending
            .Select(blog => blog.Title);
```

Set Operations

Set operations in LINQ refer to query operations that produce a result set that is based on the presence or absence of equivalent elements within the same or separate collections (or sets)

Distinct

```
var result = planets.Distinct().Select(planet => planet);

var result = from planet in planets.Distinct()
                          select planet;
```

Except

```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

var result = from planet in planets1.Except(planets2)
             select planet;

var result = planets1.Except(planets2).Select(planet => planet);
```

Intersect

```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

var result = from planet in planets1.Intersect(planets2)
             select planet;

var result = planets1.Intersect(planets2).Select(planet => planet);
```

Union

```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

var result = from planet in planets1.Union(planets2)
             select planet;

var result = planets1.Union(planets2).Select(planet => planet);
```

Where

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

var result = from word in words
             where word.Length == 3
             select word;
```

```
var result = words.Where(word => word.Length == 3).Select(word => word);
```

Quantifier Operations

Quantifier operations return a Boolean value that indicates whether some or all of the elements in a sequence satisfy a condition

All

```
class Market
{
	public string Name { get; set; }
	public string[] Items { get; set; }
}

List<Market> markets = new List<Market>
{
	new Market { Name = "Emily's"
		, Items = new string[] { "kiwi", "cheery", "banana" } },
	new Market { Name = "Kim's"
		, Items = new string[] { "melon", "mango", "olive" } },
	new Market { Name = "Adam's"
		, Items = new string[] { "kiwi", "apple", "orange" } },
};

//Determine which market have all fruit names length equal to 5

var names = from market in markets
		where market.Items.All(item => item.Length == 5)
		select market.Name;

var names = markets.Where(market => market.Items.All(item => item.Length == 5));
//Note: Below query only returns true or false. So it should be inside a where
condition
bool found = markets.All(market => market.Items.Length == 5);
```

Any

Check “All”

```
market.Items.Any(item => item.StartsWith("o"))
```

Contains

Check “All”

```
market.Items.Any(item => item.Contains("kiwi"))
```

Projection Operations

Projection refers to the operation of transforming an object into a new form that often consists only of those properties that will be subsequently used

Select

Projects values that are based on a transform function
C# Query Expression Syntax: **select**

```
List<string> words = new List<string>() { "an", "apple", "a", "day" };

var result = from word in words
             select word.Substring(0, 1);

var result = words.Select(word => word.Substring(0, 1));
```

SelectMany

- Projects sequences of values that are based on a transform function and then flattens them into one sequence
- C# Query Expression Syntax: **Use multiple from clauses**
- Select many is like cross join operation in SQL where it takes the cross product.For example if we have

```
var blogs = Blog.Blogs();
var posts = Post.Posts();

var query = blogs.SelectMany(blog => posts
      , (blog, post) => new { blog.Title, post.Content });

//Note: No key is linked. posts is independent collection
```

If Blog has a collection of Posts as property PostCollection then do the following

```
public class Blog
{
      ...
      public List<Post> PostCollection = ...
}

var query = blogs.SelectMany(blog => blog.PostCollection
      , (blog, post) => new { blog.Title, post.Content });

//Note: PostCollection is a property of blogs
```

Result is a cross join

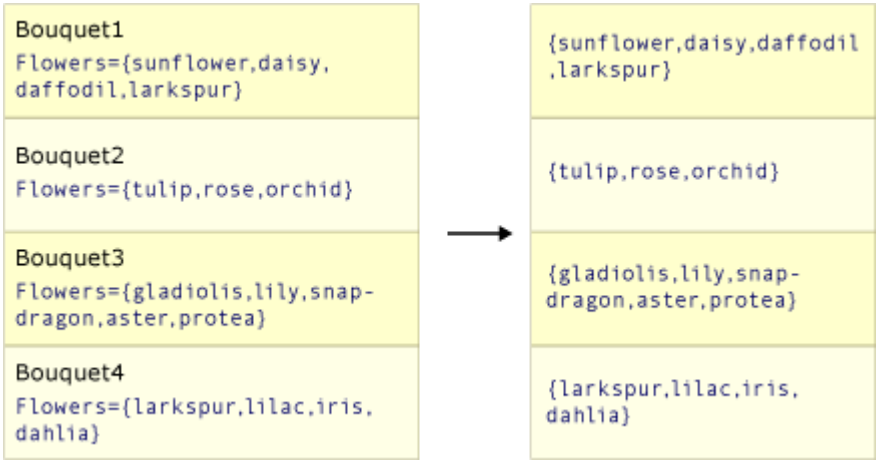
Blog1Title, Post1Content
Blog1Title, Post2Content
Blog2Title, Post1Content
Blog3Title, Post2Content

Select versus SelectMany

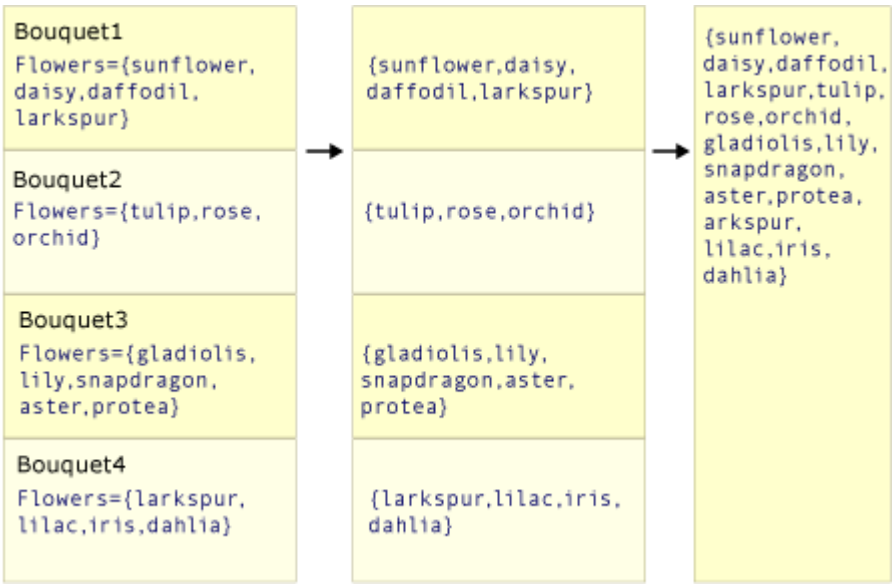
- Select() produces one result value for every source value

    - The overall result is therefore a collection that has the same number of elements as the source collection
- SelectMany() produces a single overall result that contains concatenated sub-collections from each source value

This illustration depicts how Select() returns a collection that has the same number of elements as the source collection.



This illustration depicts how SelectMany() concatenates the intermediate sequence of arrays into one final result value that contains each value from each intermediate array.




```
List<Bouquet> bouquets = new List<Bouquet>() {
       new Bouquet { Flowers = new List<string> { "sunflower", "daisy", "daffodil",
                                  "larkspur" }},
         new Bouquet{ Flowers = new List<string> { "tulip", "rose", "orchid" }},
         new Bouquet{ Flowers = new List<string> { "gladiolis", "lily",
                                               "snapdragon", "aster", "protea" }},
         new Bouquet{ Flowers = new List<string> { "larkspur", "lilac", "iris",
                                                    "dahlia" }}
};
```

```
//Select
IEnumerable<List<string>> query1 = bouquets.Select(bq => bq.Flowers);

//SelectMany
IEnumerable<string> query2 = bouquets.SelectMany(bq => bq.Flowers);
```

Group by

```
var query = posts.GroupBy(post => post.BlogID)
       .Select(post=>new { postsSub.Key, Count=postsSub.Count()});

var query = from post in posts
             group post by post.BlogID into g
             select new { g.Key, Count=g.Count() };

//Here post.Key is grouping key post => post.BlogID
```

Count

Tested way 1

```
var query = posts.GroupBy(post => post.BlogID)
                    .Select(post=>new { postsSub.Key, Count=postsSub.Count()});
```

Tested way 2

```
var blog = db.Blogs.Include(b=>b.Posts)
                    .Select(x=> new { x.BlogId, Count=x.Posts.Count()});
```

Loading Related Data

Eager loading

You can use the Include method to specify related data to be included in query results. In the following example, the blogs that are returned in the results will have their Posts property populated with the related posts

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
                    .Include(blog => blog.Posts)
}
```

Including multiple relationships

```
using (var context = new BloggingContext())
{
       var blogs = context.Blogs
                    .Include(blog => blog.Posts)
                    .Include(blog => blog.Owner)
}
```

Including multiple levels

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

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

Combine all

```
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)
}

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

Include on derived types

You can include related data from navigations defined only on a derived type using Include and ThenInclude.

```
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; }
}
```

Using cast

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

Using as operator

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

Explicit loading

```
using (var context = new BloggingContext())
{
       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();
}
```

Querying related entities

You can also get a LINQ query that represents the contents of a navigation property

```
using (var context = new BloggingContext())
{
       var blog = context.Blogs.Single(b => b.BlogId == 1);

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

You can also filter which related entities are loaded into memory

```
using (var context = new BloggingContext())
{
       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

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
```

Or when using AddDbContext

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

EF Core will then enable lazy loading for any navigation property that can be overridden--that is, it must be virtual and on a class that can be inherited from

```
public class Blog
{
    public int Id { get; set; }
    public string Name { get; set; }

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

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

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

Lazy loading without proxies

Lazy-loading proxies work by injecting the ILazyLoader service into an entity. This doesn't require entity types to be inherited from or navigation properties to be virtual, and allows entity instances created with new to lazy-load once attached to a context

```
public class Blog
{
    private ICollection<Post> _posts;
```

```
    public Blog()
    {
    }

    private Blog(ILazyLoader lazyLoader)
    {
        LazyLoader = lazyLoader;
    }

    private ILazyLoader LazyLoader { get; set; }

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

    public ICollection<Post> Posts
    {
        get => LazyLoader.Load(this, ref _posts);
        set => _posts = value;
    }
}

public class Post
{
    private Blog _blog;

    public Post()
    {
    }

    private Post(ILazyLoader lazyLoader)
    {
        LazyLoader = lazyLoader;
    }

    private ILazyLoader LazyLoader { get; set; }

    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog
    {
        get => LazyLoader.Load(this, ref _blog);
        set => _blog = value;
    }
}
```

Asynchronous Queries

- Asynchronous queries avoid blocking a thread while the query is executed in the database
- Async queries are important for keeping a responsive UI in thick-client applications

- They can also increase throughput in web applications where they free up the thread to service other requests in web applications
- **Warning :** 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. This is typically done by using the await keyword on each async operation
- Entity Framework Core provides a set of async extension methods similar to the LINQ methods, which execute a query and return results
    - **ToListAsync()**
    - **ToArrayAsync()**
    - **SingleAsync()**
- There are no async versions of some LINQ operators such as Where(...) or OrderBy(...) because these methods only build up the LINQ expression tree and don't cause the query to be executed in the database
- The EF Core async extension methods are defined in the Microsoft.EntityFrameworkCore namespace. This namespace must be imported for the methods to be available

```
public async Task<List<Blog>> GetBlogsAsync()
{
    using (var context = new BloggingContext())
    {
        return await context.Blogs.ToListAsync();
    }
}
```

Raw SQL Queries

- Raw SQL queries are useful if the query you want can't be expressed using LINQ
- Raw SQL queries are also used if using a LINQ query is resulting in an inefficient SQL query
- Raw SQL queries can **return regular entity types** or **keyless entity types** that are part of your model
- **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
- 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. Note this behavior is different from EF6. EF6 ignored property to column mapping for raw SQL queries and result set column names had to match the property names.
- 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).

Basic raw SQL queries

```
var blogs = context.Blogs
    .FromSqlRaw("SELECT * FROM dbo.Blogs")
    .ToList();

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

With parameter

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

FromSqlInterpolated is similar to FromSqlRaw but allows you to use string interpolation syntax. Just like FromSqlRaw, FromSqlInterpolated can only be used on query roots. As with the previous example, the value is converted to a DbParameter and isn't vulnerable to SQL injection.

```
var user = "johndoe";
var blogs = context.Blogs
    .FromSqlInterpolated($"EXECUTE dbo.GetMostPopularBlogsForUser {user}")
    .ToList();
```

You can also construct a DbParameter and supply it as a parameter value. Since a regular SQL parameter placeholder is used, rather than a string placeholder, FromSqlRaw can be safely used

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

FromSqlRaw allows you to use named parameters in the SQL query string, which is useful when a stored procedure has optional parameters

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

Composing with LINQ

EF Core will treat it as subquery and compose over it in the database

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

Above query generates following SQL

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

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).

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

Updating Data

EF will automatically detect changes made to an existing entity that is tracked by the context. This includes entities that you load/query from the database, and entities that were previously added and saved to the database.

Simply modify the values assigned to properties and then call SaveChanges.

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

Multiple Operations in a single SaveChanges

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

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

using (var context = new BloggingContext())
{
    // add
    context.Blogs.Add(new Blog { Url = "http://sample.com/blog_one" });
    context.Blogs.Add(new Blog { Url = "http://sample.com/blog_two" });

    // update
    var firstBlog = context.Blogs.First();
    firstBlog.Url = "";

    // remove
    var lastBlog = context.Blogs.Last();
    context.Blogs.Remove(lastBlog);

    context.SaveChanges();
}
```

Saving Related Data

```
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" },
            new Post { Title = "Intro to F#" }
        }
    };

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

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

If you change the navigation property of an entity, the corresponding changes will be made to the foreign key column in the database

```
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

By default, for required relationships, a cascade delete behavior is configured and the child/dependent entity will be deleted from the database. For optional relationships, cascade delete is not configured by default, but the foreign key property will be set to null

```
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

Allows the deletion of a row to automatically trigger the deletion of related rows

Delete behaviors

Delete behaviors are defined in the DeleteBehavior enumerator type and can be passed to the OnDelete fluent API to control whether the deletion of a principal/parent entity or the severing of the relationship to dependent/child entities should have a side effect on the dependent/child entities.

There are three actions EF can take when a principal/parent entity is deleted or the relationship to the child is severed:

- The child/dependent can be deleted
- The child's foreign key values can be set to null
- The child remains unchanged

There are four delete behaviors, as listed in the tables below

Optional relationships

Behavior Name	Effect on dependent/child in memory	Effect on dependent/child in database
Cascade	Entities are deleted	Entities are deleted
ClientSetNull (Default)	Foreign key properties are set to null	None
SetNull	Foreign key properties are set to null	Foreign key properties are set to null
Restrict	None	None

Required relationships

For required relationships (non-nullable foreign key) it is *not* possible to save a null foreign key value, which results in the following effects:

Behavior Name	Effect on dependent/child in memory	Effect on dependent/child in database
Cascade (Default)	Entities are deleted	Entities are deleted
ClientSetNull	SaveChanges throws	None
SetNull	SaveChanges throws	SaveChanges throws
Restrict	None	None

```
var blog = context.Blogs.Include(b => b.Posts).First();
var posts = blog.Posts.ToList();

context.Remove(blog);

try
{
    context.SaveChanges();
}
catch (Exception e)
{
       ...
}
```

Concurrency Conflicts (Handling)

Database concurrency refers to situations in which multiple processes or users access or change the same data in a database at the same time. Concurrency control refers to specific mechanisms used to ensure data consistency in presence of concurrent changes.

EF Core implements optimistic concurrency control, meaning that it will let multiple processes or users make changes independently without the overhead of synchronization or locking. In the ideal situation, these changes will not interfere with each other and therefore will be able to succeed. In the worst case scenario, two or more processes will attempt to make conflicting changes, and only one of them should succeed.

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.

The situation when another user has performed an operation that conflicts with the current operation is known as concurrency conflict.

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

```
using (var context = new PersonContext())
{
    // Fetch a person from database and change phone number
    var person = context.People.Single(p => p.PersonId == 1);
    person.PhoneNumber = "555-555-5555";

    // Change the person's name in the database to simulate a concurrency conflict
    context.Database.ExecuteSqlRaw(
        "UPDATE dbo.People SET FirstName = 'Jane' WHERE PersonId = 1");

    var saved = false;
    while (!saved)
    {
        try
        {
            // Attempt to save changes to the database
            context.SaveChanges();
            saved = true;
        }
        catch (DbUpdateConcurrencyException ex)
        {
            foreach (var entry in ex.Entries)
            {
                if (entry.Entity is Person)
                {
                    var proposedValues = entry.CurrentValues;
                    var databaseValues = entry.GetDatabaseValues();

                    foreach (var property in proposedValues.Properties)
                    {
                        var proposedValue = proposedValues[property];
                        var databaseValue = databaseValues[property];

                        // TODO: decide which value should be written to database
                        // proposedValues[property] = <value to be saved>;
```

```
                    }

                    // Refresh original values to bypass next concurrency check
                    entry.OriginalValues.SetValues(databaseValues);
                }
                else
                {
                    throw new NotSupportedException(
                        "Don't know how to handle concurrency conflicts for "
                        + entry.Metadata.Name);
                }
            }
        }
    }
}
```

Using Transactions

Default transaction behavior

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. For most applications, this default behavior is sufficient

Controlling transactions

```
using (var context = new BloggingContext())
{
    using (var transaction = context.Database.BeginTransaction())
    {
        try
        {
            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();

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

            // Commit transaction if all commands succeed, transaction will
auto-rollback
            // when disposed if either commands fails
            transaction.Commit();
        }
        catch (Exception)
        {
            // TODO: Handle failure
```

```
        }
    }
}
```

Asynchronous Saving

Asynchronous saving avoids blocking a thread while the changes are written to the database. This can be useful to avoid freezing the UI of a thick-client application. Asynchronous operations can also increase throughput in a web application, where the thread can be freed up to service other requests while the database operation completes

Warning: EF Core does not 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. This is typically done by using the await keyword on each asynchronous operation

```
public static async Task AddBlogAsync(string url)
{
    using (var context = new BloggingContext())
    {
        var blog = new Blog { Url = url };
        context.Blogs.Add(blog);
        await context.SaveChangesAsync();
    }
}
```

Disconnected entities

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. For example, often the request to insert a new entity is different from the request to update an existing entity.

Client identifies new entities

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. In other words, if the key has a value, then the entity was queried, sent to the client, and has now come back to be updated.

It is easy to check for an unset key when the entity type is known:

```
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:

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

Saving single entities

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

Working with graphs

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

can be inserted like this:

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

Setting Explicit Values for Generated Properties

Saving an explicit value during add

```
modelBuilder.Entity<Employee>()
       .Property(b => b.EmploymentStarted)
              .HasDefaultValueSql("CONVERT(date, GETDATE())");
```

Explicit values into SQL Server IDENTITY columns

```
using (var context = new EmployeeContext())
{
    context.Employees.Add(new Employee { EmployeeId = 100, Name = "John Doe" });
    context.Employees.Add(new Employee { EmployeeId = 101, Name = "Jane Doe" });

    context.Database.OpenConnection();
    try
    {
        context.Database.ExecuteSqlRaw("SET IDENTITY_INSERT dbo.Employees ON");
        context.SaveChanges();
        context.Database.ExecuteSqlRaw("SET IDENTITY_INSERT dbo.Employees OFF");
    }
    finally
    {
        context.Database.CloseConnection();
    }


    foreach (var employee in context.Employees)
    {
        Console.WriteLine(employee.EmployeeId + ": " + employee.Name);
    }
}
```

Pagination

```
return await _raptoroidDbContext.MyTables.Select(e => new MyTable
       {
              Id = e.Id, Name = e.Name, Age = e.Age, Weight = e.Weight,Dob = e.Dob
       }).OrderBy(e=>e.Age)
       .Skip((pageNo-1)*10).Take(pageSize)
       .ToListAsync();
```