Entity Framework Core (EF Core) is a modern object-database mapper that lets you build a clean, portable, and high-level data access layer with .NET (C#) across a variety of databases, including SQL Server (on-premises and Azure), SQLite, MySQL, PostgreSQL, Oracle, and Azure Cosmos DB.

EF Core can serve as an object-relational mapper (O/RM), which:

* Enables .NET developers to work with a database using .NET objects.
* Eliminates the need for most of the data-access code that typically needs to be written.

**The model**

With EF Core, data access is performed using a model. A model is made up of entity classes and a context object that represents a session with the database. The context object allows querying and saving data. For more information, see [Creating a Model](https://learn.microsoft.com/en-us/ef/core/modeling/).

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](https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/) to create a database from the model. Migrations allow evolving the database as the model changes.

using System.Collections.Generic;

using Microsoft.EntityFrameworkCore;

namespace Intro;

public class BloggingContext : DbContext

{

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

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(

@"Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted\_Connection=True;ConnectRetryCount=0");

}

}

public class Blog

{

public int BlogId { get; set; }

public string Url { get; set; }

public int Rating { 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; }

}

**.NET Core CLI**

* Use the following .NET Core CLI command from the operating system's command line to install or update the EF Core SQL Server provider:

.NET CLI

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

Migration Commands

Get-Migration

Drop-database

Add-Migration Migration\_Name

Update-database

Rolling back to previous migration: update-database ‘migration name’

Change Table name use – [Table(“tbl\_Genres”)]

Change Column Name – [Column(“Name”)]

After .Net 5 we have migrationbuilder.RenameColumn()

[Key] – For primary key if model doesn’t have Id property.

[Required] – Make property mandatory

EF Core Power Tools Extension- To display table relationship in VS.

1:1 relationship

**C#**

public class User

{

[Key]

public int UserId { get; set; }

public string Name { get; set; }

public Passport Passport { get; set; }

}

public class Passport

{

[Key]

[ForeignKey("User")]

public int UserId { get; set; }

public string PassportNumber { get; set; }

public User User { get; set; }

}

**Key Points**

* **Principal and Dependent Entities:** In this example, User is the principal entity, and Passport is the dependent entity.
* **Primary Key and Foreign Key:** The UserId property in the Passport entity serves as both the primary key and the foreign key.
* **Navigation Properties:** Both entities have navigation properties pointing to each other.

 Let’s create a many-to-many relationship in EF Core using a manual mapping table and data annotations. We’ll use the same Student and Course scenario.

**Step 1: Define the Entities**

First, define the Student and Course entities with navigation properties:

**C#**

public class Student

{

public int StudentId { get; set; }

public string Name { get; set; }

public List<StudentCourse> StudentCourses { get; set; } = new List<StudentCourse>();

}

public class Course

{

public int CourseId { get; set; }

public string CourseName { get; set; }

public List<StudentCourse> StudentCourses { get; set; } = new List<StudentCourse>();

}

**Step 2: Define the Join Entity**

Create a join entity StudentCourse and use data annotations to configure the relationships:

**C#**

public class StudentCourse

{

[Key, Column(Order = 0)]

public int StudentId { get; set; }

public Student Student { get; set; }

[Key, Column(Order = 1)]

public int CourseId { get; set; }

public Course Course { get; set; }

}

**Step 3: Configure the DbContext**

In your DbContext, configure the relationships using the Fluent API:

**C#**

public class SchoolContext : DbContext

{

public DbSet<Student> Students { get; set; }

public DbSet<Course> Courses { get; set; }

public DbSet<StudentCourse> StudentCourses { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<StudentCourse>()

.HasKey(sc => new { sc.StudentId, sc.CourseId });

modelBuilder.Entity<StudentCourse>()

.HasOne(sc => sc.Student)

.WithMany(s => s.StudentCourses)

.HasForeignKey(sc => sc.StudentId);

modelBuilder.Entity<StudentCourse>()

.HasOne(sc => sc.Course)

.WithMany(c => c.StudentCourses)

.HasForeignKey(sc => sc.CourseId);

}

}

AI-generated code. Review and use carefully. [More info on FAQ](https://www.bing.com/new#faq).

**Explanation**

* **Entities**: Student and Course have collections of StudentCourse to represent the many-to-many relationship.
* **Join Entity**: StudentCourse contains foreign keys StudentId and CourseId and navigation properties to Student and Course.
* **Data Annotations**: [Key, Column(Order = 0)] and [Key, Column(Order = 1)] are used to define composite keys.
* **Fluent API**: Configures the composite primary key for StudentCourse and sets up the relationships.

Fluent API is used to do advance configuration which cannot be done by data annotations.

Change Table Name, Column Name, Set Primary Key, Set IsRequired Attribute on a column

public void Configure(EntityTypeBuilder<Fluent\_BookDetail> modelBuilder)

{

//name of table

modelBuilder.ToTable("Fluent\_BookDetails");

//name of columns

modelBuilder.Property(u => u.NumberOfChapters).HasColumnName("NoOfChapters");

//primary key

modelBuilder.HasKey(u => u.BookDetail\_Id);

//other validations

modelBuilder.Property(u => u.NumberOfChapters).IsRequired();

//relations

modelBuilder.HasOne(b => b.Book).WithOne(b => b.BookDetail)

.HasForeignKey<Fluent\_BookDetail>(u => u.Book\_Id);

modelBuilder.Ignore(u => u.PriceRange); // Ignore column

modelBuilder.Property(u => u.ISBN).HasMaxLength(50);

}

1:1 mapping between Book and BookDetail on key book\_id

//relations

modelBuilder.Entity<Fluent\_BookDetail>.HasOne(b => b.Book).WithOne(b => b.BookDetail)

.HasForeignKey<Fluent\_BookDetail>(u => u.Book\_Id);

1:M mapping A publisher can have many books

modelBuilder.Entity<Fluent\_Book>.HasOne(u => u.Publisher).WithMany(u => u.Books)

.HasForeignKey(u => u.Publisher\_Id);

Context.Database.EnsureCreated() - > Check if DB and table exists before applying migration

Context.Database.getPendingMigrations().Count() – Find how many migrations are not yet applied.

Enable Logging:

options.UseSqlServer("Server=(LocalDb)\\MSSQLLocalDB;Database=CodingWiki;TrustServerCertificate=True;Trusted\_Connection=True;")

.LogTo(Console.WriteLine, new[] { DbLoggerCategory.Database.Command.Name }, LogLevel.Information);

In summary:

* **FirstOrDefault()** is generally preferred when you are dealing with potentially multiple matches.
* **SingleOrDefault()** is preferred when you expect at most one match and want to enforce that.

Ef.Functions.Like -> for Sql like operator

OrderBy -> Does Ascending

OrderByDescending-> Order by descending in LinQ

Pagination – Skip or Take

Update -> Only call dbcontext.savechanges

Delete- context.Books.Remove(book);

Eager Loading To avoid n+1 problem and it uses .include(). If property is referencing another model then it better to use .thenInclude();

Bulk Insert – using AddRange()

Bulk Delete – using RemoveRange()

Immediate Execution using – Count(),ToList(), FirstorDefault() etc

Deffered Execution- When iterating the query variable

Iqueryable inherits from Ienumerable

In Ienumerable filter is happening in the backend code side while in iqueryable we get the filtered records from the DB.

For Lazy Loading- WE need to a nuget package and configure UseLazyLoadingProxies()

Not recommended as it cause N+1 problem .

In Entity Framework (EF) Core, the ChangeTracker is a crucial component that helps track the state of entities in the context (like whether an entity is new, modified, deleted, or unchanged). It's responsible for keeping track of changes made to entities so that when SaveChanges() is called, EF Core can generate the appropriate SQL commands to update the database.

**Key Features of ChangeTracker**

* **Tracking Entity State:** It tracks the state of each entity, such as whether it has been added, modified, deleted, or is unchanged.
* **Detecting Changes:** When an entity is modified, the ChangeTracker can detect which properties have changed, so only the modified columns are updated in the database.
* **Tracking Relationships:** It tracks relationships between entities, allowing EF to handle updates to related entities as well.

**Accessing ChangeTracker**

You can access the ChangeTracker from the DbContext like this:

csharp

Copy

var changeTracker = dbContext.ChangeTracker;

**Common Operations with ChangeTracker**

1. **Getting the State of an Entity** You can get the state of an entity (whether it is Added, Modified, Deleted, or Unchanged) using the Entity() method:

csharp

Copy

var entity = dbContext.YourEntities.FirstOrDefault();

var entityState = dbContext.Entry(entity).State;

Console.WriteLine(entityState); // Output could be Added, Modified, Deleted, Unchanged

1. **Detecting Changes** EF Core automatically tracks changes made to entities in memory, but you can manually trigger change detection by calling DetectChanges():

csharp

Copy

dbContext.ChangeTracker.DetectChanges();

1. **Getting Changed Entities** You can get a list of all the entities that have been modified by checking the ChangeTracker's entries:

csharp

Copy

var modifiedEntities = dbContext.ChangeTracker.Entries()

.Where(e => e.State == EntityState.Modified)

.ToList();

1. **Accepting or Rejecting Changes** After detecting changes, you can either accept or reject changes made to tracked entities:
   * **Accept Changes:** After saving changes, EF Core resets the entity state to Unchanged to indicate that the changes have been saved.

csharp

Copy

dbContext.ChangeTracker.AcceptAllChanges();

* + **Reject Changes:** If you want to discard changes made to an entity (restore it to its original state), you can set the state to Unchanged.

csharp

Copy

dbContext.Entry(entity).State = EntityState.Unchanged;

1. **Tracking or Not Tracking Changes** By default, EF Core tracks changes for all entities, but sometimes you might not want EF to track certain entities (e.g., for performance reasons). You can disable tracking for certain queries by using .AsNoTracking():

csharp

Copy

var entities = dbContext.YourEntities.AsNoTracking().ToList();

This will prevent EF Core from tracking any changes to the entities returned by this query.

1. **Manual State Management** You can manually set the state of an entity using ChangeTracker to override EF Core's automatic tracking:

csharp

Copy

var entity = new YourEntity();

dbContext.Entry(entity).State = EntityState.Added;

1. **Detecting and Handling Deleted Entities** If you want to manually mark an entity for deletion, you can set its state to Deleted:

csharp

Copy

var entity = dbContext.YourEntities.Find(1);

dbContext.Entry(entity).State = EntityState.Deleted;

Use AsNoTracking to skip EF core to do tracking of sql queries there by improving the performance. Mostly in Get Calls for rest of the calls we need tracking

Pass Raw sql using : FromSqlRaw and FromSqlInterpolated

To execute SP use FromSqlInterpolated to execute SP with parameters