**ENTITY FRAMEWORK**

Entity framework is an ORM framework.

ORM -> Object Relational Mapping

ORM creates classes automatically based on database tables and the vice versa is also true

ORM = creates classes from database or creates database from the classes

Entity framework does all the tasks if we just provide the database schema

We have to install entity framework from NuGet -> Entity Framework

After installing the reference is also added.

**The model**

Data access performed through model

Model is made of entity classes and context objects -> represent a session within database

The model development approach supported by EF:

1. Generate a model from existing db
2. Hand code a model to match db
3. Once model is created, use EF migrations to create a database from model. Migrations allow evolving the database as model changes

**DBContext**

Lifetime : when dbcontext is created and ends when instance is disposed

Designed to use for a single unit of work ( lifetime is short usually)

The db context can be used in controllers using constructor injection

public class MyController

{

private readonly ApplicationDbContext \_context;

public MyController(ApplicationDbContext context)

{

\_context = context;

} }

DB Context pooling

When using EF Core in a web application or any scenario where multiple requests are handled concurrently, creating and disposing of a new instance of the database context for each request can be inefficient. Context pooling addresses this issue by maintaining a pool of database context instances that can be reused across multiple requests, reducing the overhead associated with creating and disposing of context instances.

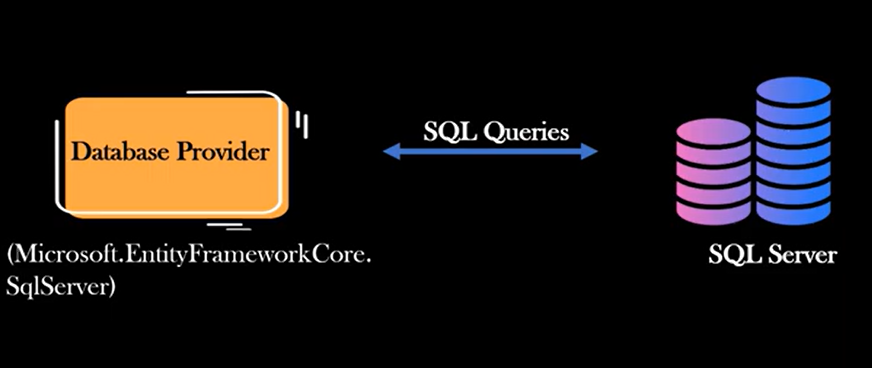
services.AddDbContextPool<ApplicationDbContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

DbSet: a class that represents an entity set in database

Public DbSet<Employee> Employees {get;set;}

What is database provider?



**Migrations:**

add-migration migration\_name

migrations will be added

update-database migration\_name // if migration name ifs not specified

advantages :

allows us to do version control our database schema changes

allows us to rollback changes if needed

allows us to apply migration without loosing existing data

**Relationships in EF Core**

1. One-to-one Relationship

Can be created using Primary key and unique foreign key

By default,EF core assumes that a property named Id or ClassNameId is primary key of an entity

Can use data annotations -> [Key]

public class Author

{

public int AuthorId { get; set; }

public string Name { get; set; }

public Biography Biography { get; set; } // Navigation property

}

public class Biography

{

[Key]

public int BiographyId { get; set; }

public string Content { get; set; }

[ForeignKey("AuthorId")]

public int AuthorId { get; set; } // Foreign key

public Author Author { get; set; } // Navigation property

}

1. One-to-many relationships

public class Author

{

public int AuthorId { get; set; }

public string Name { get; set; }

public ICollection<Book> Books { get; set; } // Navigation property for one-to-many relationship

}

public class Book

{

public int BookId { get; set; }

public string Title { get; set; }

public int AuthorId { get; set; } // Foreign key

[ForeignKey("AuthorId")]

public Author Author { get; set; } // Navigation property for the associated author

}

1. Many-to-many

public class Student

{

public int StudentId { get; set; }

public string Name { get; set; }

public ICollection<StudentCourse> StudentCourses { get; set; } // Navigation property for many-to-many relationship

}

public class Course

{

public int CourseId { get; set; }

public string Title { get; set; }

public ICollection<StudentCourse> StudentCourses { get; set; } // Navigation property for many-to-many relationship

}

public class StudentCourse

{

[Key]

public int StudentId { get; set; }

public Student Student { get; set; }

[Key]

public int CourseId { get; set; }

public Course Course { get; set; }

}

**Fluent API**

to configure entity mappings and relationships using method chaining and lambda expressions

The Fluent API configuration typically takes place inside the OnModelCreating method of your DbContext class. This method is called by EF Core when it's initializing the model.

For each entity in your model, you can use Fluent API methods to configure various aspects such as primary keys, indexes, column mappings, relationships, etc.

allows you to configure relationships between entities in a more detailed manner compared to data annotations. You can specify the foreign key properties, navigation properties, cardinality, cascade behavior, and more.

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

// Configure Student entity

modelBuilder.Entity<Student>()

.HasKey(s => s.StudentId);

// Configure Course entity

modelBuilder.Entity<Course>()

.HasKey(c => c.CourseId);

// Configure StudentCourse entity (join table)

modelBuilder.Entity<StudentCourse>()

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

// Configure one-to-many relationship between Student and StudentCourse

modelBuilder.Entity<StudentCourse>()

.HasOne(sc => sc.Student)

.WithMany(s => s.StudentCourses)

.HasForeignKey(sc => sc.StudentId);

// Configure one-to-many relationship between Course and StudentCourse

modelBuilder.Entity<StudentCourse>()

.HasOne(sc => sc.Course)

.WithMany(c => c.StudentCourses)

.HasForeignKey(sc => sc.CourseId);

}

1. One-to One

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Author>()

.HasOne(a => a.Biography)

.WithOne(b => b.Author)

.HasForeignKey<Biography>(b => b.AuthorId);

}

1. One-to-many

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Course>()

.HasMany(c => c.Students)

.WithOne(s => s.Course)

.HasForeignKey(s => s.CourseId);

}

1. Many-to-one

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Student>()

.HasOne(s => s.School)

.WithMany(school => school.Students)

.HasForeignKey(s => s.SchoolId);

}

1. Many-to-many

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);

}

**Adding data into database**

Create instances of model classes

Add them to appropriate DbSet property in your DbContext class

Call the save changes

// Step 1: Define entity classes representing tables in the database

public class Student

{

public int StudentId { get; set; }

public string Name { get; set; }

}

// Step 2: Instantiate DbContext

public class SchoolDbContext : DbContext

{

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

// Configure DbContext to use your database provider and connection string

optionsBuilder.UseSqlServer("your\_connection\_string\_here");

}

}

// Step 3: Create entity instances and populate them with data

Student newStudent = new Student { Name = "John Doe" };

// Step 4: Add entities to DbContext

using (var dbContext = new SchoolDbContext())

{

dbContext.Students.Add(newStudent);

dbContext.SaveChanges();

}

**Retrieve data using EF core**

var student=appDbContext.Student.ToList(); // getting data

var stu=appDbContext.Student.Where(x=>x.id==id).FirstOrDefault() //gettimg student data with id

**Updating dating using EF core**

1. Retrieve data to update

var stu=appDbContext.Student.Where(x=>x.id==id).FirstOrDefault()

1. Specify the changes

Stu.Name=”Nikhita”;

1. Save the changes

appDbContext.SaveChanges();

**Deleting data**

1. Retrieve data to delete

Var stu= appDbContext.Student.Where(x=>x.id==id).FirstOrDefault()

1. Delete the data

appdbContext.Student.Remove(stu)

1. Save the changes

appDbContext.SaveChanges()

**Eager Loading**

To load related data we have :

1. Eager loading
2. Lazy loading
3. Explicit loading

Advantages of eager loading

It can reduce the number of Database round-trips required to retrieve related data

We can fetch them all at a time

Disadvantages

This may result in a larger amount of data being transferred between the database and the application

Over-fetching of data

var booksWithAuthors = context.Books

.Include(b => b.Author) // Eager loading Author navigation property

.ToList();

var booksWithAuthorsAndPublishers = context.Books

.Include(b => b.Author) // Eager loading Author navigation property

.ThenInclude(a => a.Publishers) // Eager loading Publishers navigation property of Author

.ToList();

**Explicit loading**

Related data is explicitly loaded from database at a later time

We use Entry method of DbContext class along with Collection or Reference methods to explicitly load related entities

var book = context.Books.FirstOrDefault(b => b.BookId == 1);

// Explicitly load the author of the book

if (book != null)

{

context.Entry(book).Reference(b => b.Author).Load();

}

// Now you can access the author of the book

var author = book?.Author;

Lazy Loading

that automatically loads related entities from the database when they are accessed for the first time. It allows you to navigate through navigation properties of entities without explicitly loading related data, as EF Core transparently retrieves it from the database on demand.

1. When you access a navigation property of an entity, EF Core checks if the related data has already been loaded.
2. If the related data has not been loaded yet, EF Core automatically generates and executes a query to retrieve the related data from the database.
3. The retrieved related entities are then attached to the navigation property, and subsequent accesses to that navigation property will return the already loaded data.

To enable lazy loading in EF Core, you need to install the Microsoft.EntityFrameworkCore.Proxies package and enable it for your DbContext by configuring the context options.

Can enable it in DbContext like:

public class YourDbContext : DbContext

{

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

// Enable lazy loading

optionsBuilder.UseLazyLoadingProxies().UseSqlServer("your\_connection\_string");

}

public DbSet<Book> Books { get; set; }

public DbSet<Author> Authors { get; set; }

}

With lazy loading enabled, when you access book.Author for the first time, EF Core will automatically load the related Author entity from the database. Similarly, accessing author.Books will trigger a query to load the related books for that author. Lazy loading can simplify your code by allowing you to work with related entities without worrying about loading them explicitly. However, it's essential to be aware of the potential performance implications, especially in scenarios with a large number of database round-trips.

**Asynchronous Operations in entity framework**

In synchronous programming, the program waits for the operation to finish before moving on to the next line of code

But, the problem here is the program is idle and waits for operation to complete

This approach is a problem when we have long-running or I/O-bound operations

So, to avoid these problems we have asunchronous programs and EF core provides asynchronous versions of most database operations

When calling asynchronous methods, we use async keyword

We use await keyword to asynchronously wait for completion of the operation

Example : AddAsync, ToListAsync

**Data Annotations**

Attributes that can be applied to model classes to define various behaviors

Used for :

1. Defining validation rules
2. Data formatting
3. Localisation
4. Database schema generation

We can also fluent api to customize the model, if both are used then fluent api will override the configuration performed data annotations

**Table Attribute**

Tells the name of the table associated with entity class

[Table("Books")]

public class Book

{

// Properties

}

**KeyAttribute**

To specify the primary key

public class Book

{

[Key]

public int Id { get; set; }

}

**ColumnAttribute**

Specifies name, data type, and other attributes of a column in db table

public class Book

{

[Column("BookTitle")]

public string Title { get; set; }

}

**RequiredAttribute**

Specifies that property is required and cant be null

public class Book

{

[Required]

public string Title { get; set; }

}

**MaxLength**

Specifies the maximum length of string property

public class Book

{

[MaxLength(100)]

public string Title { get; set; }

}

**Foreign key**

Specifies foreign key property for dependent entity

public class Book

{

public int AuthorId { get; set; }

[ForeignKey("AuthorId")]

public Author Author { get; set; }

}

**Change Tracking**

Mechanism that keeps track of changes made to entities within context

Ef core automatically detects changes made to entites and manges the persistence of these changes to underlying database

How Change tracking works?

* Db context has ChangeTracker component that tracks changes made to entities
* changeTracker monitors the state of each entity within the context and assigns an EntityState to each entity, such as **Added, modified, deleted or unchanged** based on the changes made to entity
* When we call SaveChanges method, Ef core examines the tracked entities and generates the necessary SQL statements based on current states
* After SaveChanges is called, changetracker resets the entity states to unchanged as the changes have been successfully persisted to database
* Different states of entities

1. Detached : entities not tracked by DbContext
2. Added : entity is new and have not yet been inserted to db. Inserted once SaveChanges is called
3. UnChanged : entities have not been changed since they wre queried from the database. All entities initially will be in this state
4. Modified : entity is modified and have not been changed in db. They will be updated once the SaveChanges is called
5. Deleted : entities exist in db, but are marked to delete and will be deleted once savechanges is called