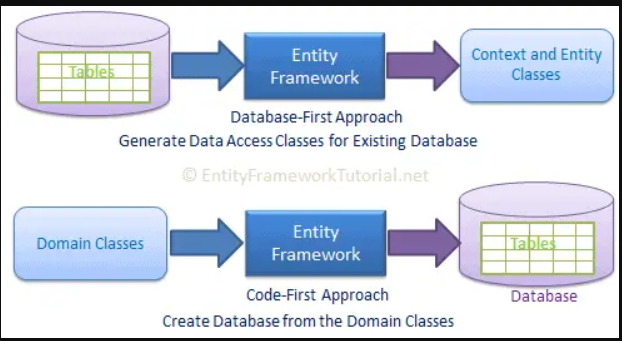
# Entity Framework Core:

Entity Framework (EF) Core is a lightweight, extensible, open source and cross-platform version of the popular Entity Framework data access technology.

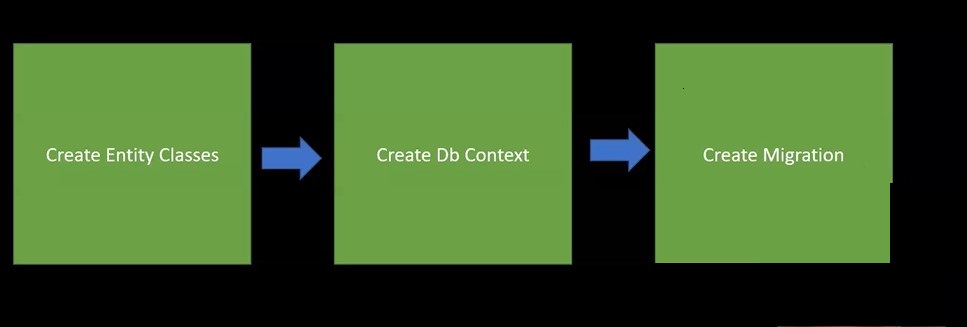
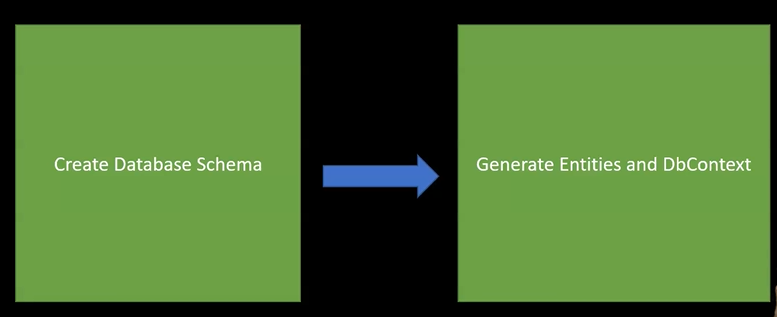
EF Core can serve as an Object-Relational Mapper (ORM), enabling .NET developers to work with a database using .NET objects, and eliminating the need for most of the data-access code they usually need to write.



Entity Framework Core is a modern object-database mapper for .NET. It simplifies working with various databases (including Azure Cosmos DB, MySQL, PostgreSQL, SQL Server, and SQLite) using strongly-typed .NET objects and support for LINQ.

Reduces the code needed for data access.



* Creates Database from code(Code first).
* Generates Entities and DbContext from Database (Db first)

Using EF stages:

* Install EF nuget packages
* Create Models
* Create Context
* Create migration
* Apply CRUD methods directly or through repositories
* Save Changes

**Installation:**

**Nuget:**

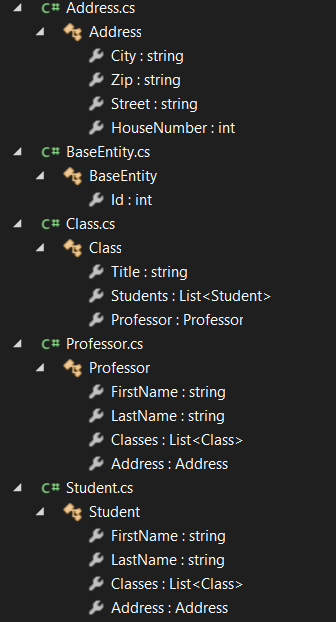
* Microsoft.EntityFrameworkCore.Sqlite/SqlServer;
* Microsoft.EntityFrameworkCore.Tools
* Microsoft.EntityFrameworkCore.Design
* Microsoft.EntityFrameworkCore.Proxies
* [SqlLite browser](https://sqlitebrowser.org/dl/)

**Models:**

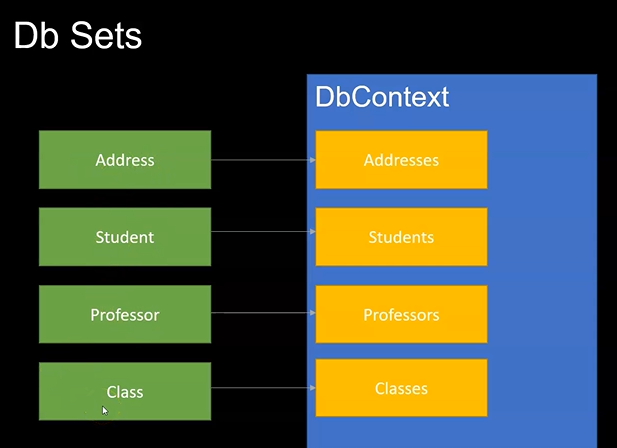
First we create Models that would be mapped to Tables.

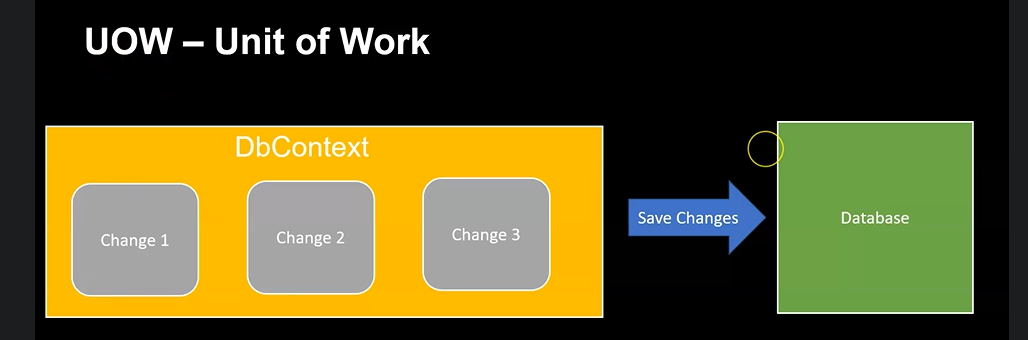
All POCO Entities should inherit from base Entity.

A POCO Entities (Plain Old CLR Object) is a class that does not depend on any framework-specific base class. It is like any other normal . NET CLR class; that is why it is called "Plain Old CLR Objects". POCO entities are supported in both EF6 and EF Core.



**DbContext:**

Define a class that derives from DbContext and exposes DbSet properties that represent collections of the specified entities in the context. If you are working with the EF Designer, the context will be generated for you. If you are working with Code First, you will typically write the context yourself.



public class ApplicationDbContext : DbContext

{

public DbSet<Address> Addresses { get; set; }

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

public DbSet<Professor> Professors { get; set; }

public DbSet<Class> Classes { get; set; }

//To work with sqlite

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlite("Filename=EfCoreAcademy.db");

base.OnConfiguring(optionsBuilder);

}

**protected readonly DbContextOptions Configuration;**

**public AppDbContext(DbContextOptions configuration) : base(configuration)**

**{**

**Configuration = configuration;**

**}**

//Defines tables relations

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Address>().HasKey(e => e.Id);

modelBuilder.Entity<Student>().HasKey(e => e.Id);

modelBuilder.Entity<Professor>().HasKey(e => e.Id);

modelBuilder.Entity<Class>().HasKey(e => e.Id);

modelBuilder.Entity<Student>().HasOne(e => e.Address);

modelBuilder.Entity<Professor>().HasOne(e => e.Address);

modelBuilder.Entity<Class>().HasMany(e => e.Students).WithMany(e => e.Classes);

modelBuilder.Entity<Class>().HasOne(e => e.Professor).WithMany(e => e.Classes);

base.OnModelCreating(modelBuilder);

}

}

In this Example we use Fluent Api.

Most of operations can be done using Attributes as:

[Key]

[Required]

[MaxLength(24)]

[MinLength(24)]

[Table("StudentsInfo")]

[Column("FirstName")]

[ForeignKey("StudentID")]

[NotMapped]

Create the Context in the program.cs:

Sqlite:

var options = new DbContextOptionsBuilder<ApplicationDbContext>().UseSqlite().Options;

var dbContext = new ApplicationDbContext(options);

dbContext.Database.Migrate();

sql Server:

builder.Services.AddDbContext<AppDbContext>(options => options.UseSqlServer(builder.Configuration.GetConnectionString("DbConnection")));

EF8:

Property to be lazy loaded should be marked as Virtual:

builder.Services.AddDbContext<AppDbContext>(options =>

options.UseLazyLoadingProxies().UseSqlServer(builder.Configuration.GetConnectionString("DbConnection")));

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

setting.json:

"ConnectionStrings": {

"DbConnection": "data source=TallyPC\\MSSQLSERVER01; initial catalog=DBName;trusted\_connection=true ;Encrypt=False "

},

Database Creation:

View -> other windows->package manger console

1. Generate Migration:

In the CLI change to project folder

CLI: dotnet ef migrations add InitialMigration

PM: Add-migration MigrationName



The Db file will be placed in the project folder

The code to use it:

dbContext.SaveChanges();

**Dbset Methods:**

1. Add:

dbContext = new ApplicationDbContext(options);

var address = new Address() { City = "Hamburg", Street = "Demostreet", Zip = "24225", HouseNumber = 1 };

var professor = new Professor() { FirstName = "Jonathan", LastName = "Schoolman", Address = address };

var student1 = new Student() { FirstName = "John", LastName = "Doe", Address = address };

var student2 = new Student() { FirstName = "Maria", LastName ="Maker", Address = address };

var class1 = new Class() { Professor = professor, Students = new List<Student> { student1, student2 }, Title = "IT" };

dbContext.Addresses.Add(address);

dbContext.Students.Add(student1);

dbContext.Students.Add(student2);

dbContext.Professors.Add(professor);

dbContext.Classes.Add(class1);

dbContext.SaveChanges();

dbContext.Dispose();

1. Delete:

dbContext.Professors.RemoveRange(professors);

void ProcessDelete()

{

var professors = dbContext.Professors.ToList();

var students = dbContext.Students.ToList();

var classes = dbContext.Classes.ToList();

var addresses = dbContext.Addresses.ToList();

dbContext.Professors.RemoveRange(professors);

dbContext.Students.RemoveRange(students);

dbContext.Classes.RemoveRange(classes);

dbContext.Addresses.RemoveRange(addresses);

dbContext.SaveChanges();

dbContext.Dispose();

}

1. Get: var students = dbContext.Students.Include(s => s.Classes).Where (s => s.FirstName == "Maria").ToList();

void ProcessSelect()

{

dbContext = new ApplicationDbContext(options);

//var professor = dbContext.Professors.Include(p => p.Address).Single(p => p.FirstName == "Jonathan");

var student = dbContext.Students.Include(s => s.Classes).Where(s => s.FirstName == "Maria").ToList();

dbContext.Dispose();

}

1. Update:

var student = dbContext.Students.Include(s => s.Classes).First();

student.FirstName = "Tim";

student.Classes = new List<Class>();

dbContext.SaveChanges();

**Exercise:**

Create Students Controller with all CRUD Methods

# Ef inheritance:

# **Table-Per-Hierarchy (TPH)**

# TPH is the default inheritance mapping pattern in EF Core. -It uses a single table to store the data for all types in the hierarchy, and a discriminator column is used to identify the type of each row. -It is the most efficient inheritance strategy in terms of query performance because only one table needs to be queried.

modelBuilder.Entity<Blog>().HasDiscriminator<string>("blog\_type") .HasValue<Blog>("blog\_base") .HasValue<RssBlog>("blog\_rss");

**Table-per-type(TPT):**  
In the TPT mapping pattern, all the types are mapped to individual tables. Properties that belong solely to a base type or derived type are stored in a table that maps to that type. Tables that map to derived types also store a foreign key that joins the derived table with the base table.  
modelBuilder.Entity<Blog>().UseTptMappingStrategy()  
  
TPT is a useful mapping strategy when subtypes have very different sets of properties, as it allows the columns in each table to correspond exactly to the properties of each subtype.  
  
**Table per class/concrete type(TPC):**  
In the TPC mapping pattern, all the types are mapped to individual tables. Each table contains columns for all properties on the corresponding entity type. This addresses some common performance issues with the TPT strategy.

modelBuilder.Entity<Blog>().UseTpcMappingStrategy()

    .ToTable("Blogs");

modelBuilder.Entity<RssBlog>()

    .ToTable("RssBlogs");

**Benefits of TPC**

**Normalization:** Reduces duplication of data and maintains consistency.

**Performance:** Queries involving concrete types are faster as they avoid joins.

**Flexibility:** Each table can evolve independently in terms of schema changes.

# EF and Lazy loading:

# **How does lazy loading work in EF Core?**

In EF Core, lazy loading is achieved through the use of proxy objects. When an entity is loaded from the database, a proxy object is created in place of the real entity. When a property or navigation property of the entity is accessed, the proxy object intercepts the access and loads the related data from the database.

## protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

## {

## optionsBuilder.UseLazyLoadingProxies();

## }

public virtual List<Book> Books { get; set; } = new List<Book>();

**Eager loading:**

var person = \_context.Person.Include(p=> p.Hats).Where(p=> p.id == id).ToList();

Lazy loading can cause unneeded extra database roundtrips to occur

# Unit of work + Repository design pattern:



IstudentRepository :Create repository interface with Required CRUD methods.

public class StudentRepository : IStudentRepository {

private SchoolContext context;

public StudentRepository(SchoolContext context)

{

this.context = context;

}

public IEnumerable<Student> GetStudents()

{

return context.Students.ToList();

}

public Student GetStudentByID(int id)

{

return context.Students.Find(id);

}

public void InsertStudent(Student student)

{

context.Students.Add(student);

Save();

}

public void DeleteStudent(int studentID)

{

Student student = context.Students.Find(studentID);

context.Students.Remove(student);

Save();

}

public void UpdateStudent(Student student)

{

context.Entry(student).State = EntityState.Modified;

Save();

}

public void Save()

{

context.SaveChanges();

}

}

studentRepository Implement repository interface with Required CRUD methods.

You could instantiate a new context in the repository, but then if you used multiple repositories in one controller, each would end up with a separate context. Later you'll use multiple repositories in the Course controller, and you'll see how a unit of work class can ensure that all repositories use the same context.

Implement a Generic Repository and a Unit of Work Class

Creating a repository class for each entity type could result in a lot of redundant code, and it could result in partial updates. For example, suppose you have to update two different entity types as part of the same transaction. If each uses a separate database context instance, one might succeed and the other might fail. One way to minimize redundant code is to use a generic repository, and one way to ensure that all repositories use the same database context (and thus coordinate all updates) is to use a unit of work class.

public class GenericRepository<T>: IGenericRepository<T> where T: class

{

private EmployeeDBContext \_context = null;

//The following Variable is going to hold the DbSet Entity

private DbSet<T> table = null;

//Using the Parameterless Constructor,

//we are initializing the context object and table variable

public GenericRepository()

{

this.context = new EmployeeDBContext();

//Whatever class name we specify while creating the instance of GenericRepository

//That class name will be stored in the table variable

table = \_context.Set<T>();

}

//Using the Parameterized Constructor,

//we are initializing the context object and table variable

public GenericRepository(EmployeeDBContext \_context)

{

this.\_context = \_context;

table = \_context.Set<T>();

}

//This method will return all the Records from the table

public IEnumerable<T> GetAll()

{

return table.ToList();

}

//This method will return the specified record from the table

//based on the ID which it received as an argument

public T GetById(object id)

{

return table.Find(id);

}

//This method will Insert one object into the table

//It will receive the object as an argument which needs to be //inserted into the database

public void Insert(T obj)

{

//It will mark the Entity state as Added State

table.Add(obj);

}

}

}

//This method is going to update the record in the table

//It will receive the object as an argument

public void Update(T obj)

{

//First attach the object to the table

table.Attach(obj);

//Then set the state of the Entity as Modified

\_context.Entry(obj).State = EntityState.Modified;

}

//This method is going to remove the record from the table

//It will receive the primary key value as an argument whose information //needs to be removed from the table

public void Delete(object id)

{

//First, fetch the record from the table

T existing = table.Find(id);

//This will mark the Entity State as Deleted

table.Remove(existing);

}

public virtual void Delete(TEntity entityToDelete)

{

if (context.Entry(entityToDelete).State == EntityState.Detached)

{

dbSet.Attach(entityToDelete);

}

dbSet.Remove(entityToDelete);

}

public virtual void Update(TEntity entityToUpdate)

{

dbSet.Attach(entityToUpdate);

context.Entry(entityToUpdate).State = EntityState.Modified;

}

//This method will make the changes permanent in the database

//That means once we call Insert, Update, and Delete Methods,

//Then we need to call the Save method to make the changes permanent in the database

public void Save()

{

\_context.SaveChanges();

}

}

# Async Generic Repository:

public class AsyncGenericRepository<T>: IAsyncGenericRepository<T> where T : BaseEntity

{

//The following variable is going to hold the EmployeeDBContext instance

private AppDbContext \_context = null;

//The following Variable is going to hold the DbSet Entity

private DbSet<T> table = null;

public AsyncGenericRepository(AppDbContext \_context)

{

this.\_context = \_context;

table = \_context.Set<T>();

}

public Task<List<T>> GetAllAsync()

{

return table.ToListAsync();

}

public Task<T> GetByIdAsync(object id)

{

return Task.FromResult(table.Find(id));

}

public Task UpdateAsync(T entity)

{

table.Update(entity);

return Task.CompletedTask;

}

public Task DeleteAsync(T entity)

{

table.Remove(entity);

return Task.CompletedTask;

}

public Task DeleteAsync(int id)

{

var entityToDelete = table.FirstOrDefault(x => x.Id == id);

if (entityToDelete != null)

{

this.table.Remove(entityToDelete);

return Task.CompletedTask;

}

else

{

return null;

}

}

public async Task<T> InsertAsync(T entity)

{

await table.AddAsync(entity);

return entity;

}

}

# Use:

[Route("api/[controller]")]

[ApiController]

public class StudentController : ControllerBase

{

AppDbContext \_context;

public ValuesController(AppDbContext context)

{

\_context = context;

}

[HttpGet(Name = nameof(GetAllProducts))]

[ProducesResponseType(200, Type = typeof(List<Product>))]

public async Task<IActionResult> GetAllProducts()

{

IAsyncGenericRepository<Student> repo = new

AsyncGenericRepository<Student>(\_context);

var result = await repo.GetAllAsync();

return this.Ok(result);

}

[HttpGet("{id}")]

public async Task<IActionResult> GetById(int id)

{

IAsyncGenericRepository<Student>repo = new

AsyncGenericRepository<Student>(\_context);

var result = await repo.GetByIdAsync(id);

if (result == null)

{

return NotFound();

}

return Ok(result);

}

[HttpPost(Name = nameof(AddNewStudent))]

[ProducesResponseType(200, Type = typeof(Student))]

public async Task<ActionResult<Product>> AddNewStudent(Student s)

{

IAsyncGenericRepository<Student>repo = new

AsyncGenericRepository<Student>(\_context);

var newProductAfterInsert = await repo.InsertAsync(s);

await \_context.SaveChangesAsync();

return Created("products/" + newProductAfterInsert.Id, newProductAfterInsert);

}

[HttpDelete("{id}", Name = nameof(DeleteProduct))]

public async Task<IActionResult> DeleteStudent(int id)

{

IAsyncGenericRepository<Student>repo = new

AsyncGenericRepository<Student>(\_context);

await repo.DeleteAsync(id);

await repo.SaveChangesAsync();

return NoContent();

}

}

# Creating the Unit of Work Class

The unit of work pattern uses a single transaction or a single unit of work for multiple insert, update, and delete operations. These operations either succeed or failure as an entire unit. In other words, all of the operations will be committed as one transaction or rolled back as a single unit.

The unit of work class serves one purpose: to make sure that when you use multiple repositories, they share a single database context. That way, when a unit of work is complete you can call the SaveChanges method on that instance of the context and be assured that all related changes will be coordinated. All that the class needs is a Save method and a property for each repository. Each repository property returns a repository instance that has been instantiated using the same database context instance as the other repository instances.

Each repository property checks whether the repository already exists. If not, it instantiates the repository, passing in the context instance. As a result, all repositories share the same context instance.

UOW would be created as a controller member and disposed as the controller disposed.

public class UnitOfWork

{

private SchoolContext context;

private GenericRepository<Department> departmentRepository;

private GenericRepository<Course> courseRepository;

public UnitOfWork(SchoolContext context)

{

Context = context;  
 }

public GenericRepository<Department> DepartmentRepository

{

get

{

if (this.departmentRepository == null)

{

this.departmentRepository = new GenericRepository<Department>(context);

}

return departmentRepository;

}

}

public GenericRepository<Course> CourseRepository

{

get

{

if (this.courseRepository == null)

{

this.courseRepository = new GenericRepository<Course>(context);

}

return courseRepository;

}

}

public void Save()

{

context.SaveChanges();

}

}

}

public void SaveUserInfo(User user)

{

using (var uow = new UnitOfWork())

{

var dbUser = uow.CourseRepository.GetByKey(user.Id);

if (dbUser == null)

{

// TODO:

}

dbUser.Name = user.Name;

dbUser.Address = user.Address;

// Or use a framework for injecting properties

uow.SaveAndAcceptChanges();

}

}