Entity Framework Core

A Modern ORM for .NET Applications

Press Space for next page →

Introduction to Entity Framework

Entity Framework (EF) Core is a lightweight, extensible, open-source, and cross-platform ORM (Object-Relational Mapper) for .NET.

What is an ORM?

An Object-Relational Mapper bridges the gap between your object-oriented code and relational databases, allowing you to work with databases using C# objects instead of writing SQL.

Key Benefits

- ✅ Reduces boilerplate ADO.NET code
- ✅ Type-safe database operations
- ✅ Automatic SQL generation
- ✅ Cross-database compatibility

EF Core Key Features

What makes Entity Framework Core powerful

- 🎯 **LINQ Support** - Write type-safe queries using LINQ
- 🔄 **Change Tracking** - Automatic detection of entity changes
- 💾 **Multiple Providers** - SQL Server, PostgreSQL, SQLite, In-Memory, and more
- 🧪 **Testing Support** - In-memory database for unit testing
- 🔧 **Migrations** - Code-first database schema management
- ⚡ **Performance** - Optimized queries and connection pooling

EF Core and LINQ

Entity Framework leverages **LINQ** (Language Integrated Query) for querying databases

Traditional SQL vs LINQ

Basic Query Examples

Common query patterns with Entity Framework Core

```
// Basic select all
var allProducts = await context.Products.ToListAsync();

// Filtering with Where
var expensiveProducts = await context.Products
    .Where(p => p.Price > 100)
    .ToListAsync();

// Ordering results
var sortedProducts = await context.Products
    .OrderBy(p => p.Name)
    .ThenByDescending(p => p.Price)
    .ToListAsync();

// Projecting to anonymous types
var productSummaries = await context.Products
    .Select(p => new {
        p.Name,
        p.Price,
        DiscountedPrice = p.Price * 0.9m
    })
    .ToListAsync();
```

More Query Examples

Advanced querying techniques

Insert Examples

Adding new entities to the database

```
// Create and add a single product
var product = new Product
{
    Name = "Gaming Laptop",
    Price = 1299.99m,
    InStock = true,
    Category = "Electronics"
};

context.Products.Add(product);
await context.SaveChangesAsync(); // Commits to database
// product.Id is now populated with database-generated ID

// Add multiple entities at once
var products = new List<Product>
{
    new Product { Name = "Mouse", Price = 29.99m },
    new Product { Name = "Keyboard", Price = 79.99m },
    new Product { Name = "Monitor", Price = 299.99m }
};

context.Products.AddRange(products);
await context.SaveChangesAsync();
```

Update Examples

Modifying existing entities

TRACKED ENTITY UPDATE

```
// Retrieve entity (automatically tracked by context)
var product = await context.Products
    .FirstAsync(p => p.Id == 1);

// Modify properties
product.Price = 899.99m;
product.InStock = true;

// SaveChanges automatically detects changes
await context.SaveChangesAsync();
```

UNTRACKED UPDATE

```
// Update without loading from database first
var product = new Product
{
    Id = 1,
    Name = "Updated Name",
    Price = 799.99m,
    InStock = true
};
```

More Update Examples

Advanced update techniques

PARTIAL UPDATE

```
// Update only specific properties
var product = await context.Products
    .FirstAsync(p => p.Id == 1);

// Update specific property only
context.Entry(product)
    .Property(p => p.Price)
    .CurrentValue = 699.99m;

await context.SaveChangesAsync();
```

BULK UPDATE (EF CORE 7+)

```
// Update multiple records without loading them
// No change tracking overhead - executes directly on database
await context.Products
    .Where(p => p.Category == "Electronics")
    .ExecuteUpdateAsync(
        s => s.SetProperty(
            p => p.Price,
            p => p.Price * 0.9m  // 10% discount
```

Delete Examples

Removing entities from the database

```
// Delete tracked entity
var product = await context.Products
    .FirstAsync(p => p.Id == 1);

context.Products.Remove(product);
await context.SaveChangesAsync();

// Delete without loading (more efficient)
var product = new Product { Id = 1 };
context.Products.Remove(product);
await context.SaveChangesAsync();

// Delete multiple entities
var oldProducts = await context.Products
    .Where(p => p.Price < 10)
    .ToListAsync();

context.Products.RemoveRange(oldProducts);
await context.SaveChangesAsync();

// Bulk Delete (EF Core 7+) - Most efficient
await context.Products
```

Repository Pattern

Abstraction layer for data access logic

Why Use Repository Pattern?

- ✅ Separates data access from business logic
- ✅ Makes code more testable (easier to mock)
- ✅ Centralizes query logic
- ✅ Enables dependency injection
- ✅ Provides consistent API across entities

When to Use?

- Complex applications with multiple data sources
- When you need to swap implementations (testing, different databases)
- Team projects requiring consistent data access patterns

Repository Interface

Defining the contract for data access

```
public interface IRepository<T> where T : class
{
    // Query operations
    Task<T?> GetByIdAsync(int id);
    Task<IEnumerable<T>> GetAllAsync();
    Task<IEnumerable<T>> FindAsync(Expression<Func<T, bool>> predicate);

    // Command operations
    Task AddAsync(T entity);
    Task AddRangeAsync(IEnumerable<T> entities);
    void Update(T entity);
    void Remove(T entity);
    void RemoveRange(IEnumerable<T> entities);

    // Persistence
    Task<int> SaveChangesAsync();
}
```

This interface can be used with any entity type in your application

Repository Implementation

Generic repository implementation

```
public class Repository<T> : IRepository<T> where T : class
{
    protected readonly DbContext _context;
    protected readonly DbSet<T> _dbSet;

    public Repository(DbContext context)
    {
        _context = context;
        _dbSet = context.Set<T>();
    }

    public virtual async Task<T?> GetByIdAsync(int id)
        => await _dbSet.FindAsync(id);

    public virtual async Task<IEnumerable<T>> GetAllAsync()
        => await _dbSet.ToListAsync();

    public virtual async Task<IEnumerable<T>> FindAsync(
        Expression<Func<T, bool>> predicate)
        => await _dbSet.Where(predicate).ToListAsync();

    public virtual async Task AddAsync(T entity)
        => await _dbSet.AddAsync(entity);
```

Unit of Work Pattern

Coordinating multiple repositories in a single transaction

What is Unit of Work?

A pattern that maintains a list of objects affected by a business transaction and coordinates the writing of changes and resolution of concurrency problems.

Key Benefits

- ✅ Groups multiple operations into a single transaction
- ✅ Ensures data consistency across repositories
- ✅ Single point of save/commit for all changes
- ✅ Simplifies transaction management
- ✅ Reduces database round trips

Unit of Work Interface

Defining the contract for coordinated data access

```
public interface IUnitOfWork : IDisposable
{
    // Repository properties - access to all repositories
    IRepository<Product> Products { get; }
    IRepository<Category> Categories { get; }
    IRepository<Order> Orders { get; }
    IRepository<Customer> Customers { get; }

    // Persistence operations
    Task<int> SaveChangesAsync();
}
```

Key Concepts:

- Single interface provides access to multiple repositories
- `SaveChangesAsync()` commits all pending changes across all repositories
- Implements `IDisposable` for proper resource cleanup

Unit of Work Transactions

Adding transaction support to Unit of Work

```
public interface IUnitOfWork : IDisposable
{
    IRepository<Product> Products { get; }
    IRepository<Category> Categories { get; }
    IRepository<Order> Orders { get; }

    Task<int> SaveChangesAsync();

    // Explicit transaction management
    Task BeginTransactionAsync();
    Task CommitTransactionAsync();
    Task RollbackTransactionAsync();
}
```

Transaction Methods:

- `BeginTransactionAsync()` - Start a new database transaction
- `CommitTransactionAsync()` - Commit all changes permanently
- `RollbackTransactionAsync()` - Undo all changes in the transaction

Unit of Work Implementation

Creating the implementation class

```
public class UnitOfWork : IUnitOfWork
{
    private readonly AppDbContext _context;
    private IDbContextTransaction? _transaction;

    public UnitOfWork(AppDbContext context)
    {
        _context = context;
        Products = new Repository<Product>(_context);
        Categories = new Repository<Category>(_context);
        Orders = new Repository<Order>(_context);
    }

    public IRepository<Product> Products { get; }
    public IRepository<Category> Categories { get; }
    public IRepository<Order> Orders { get; }

    public async Task<int> SaveChangesAsync()
        => await _context.SaveChangesAsync();

    public async Task BeginTransactionAsync()
        => _transaction = await _context.Database.BeginTransactionAsync();
```

Unit of Work Usage Example

Practical example with transaction management

```
public class ProductService
{
    private readonly IUnitOfWork _unitOfWork;

    public ProductService(IUnitOfWork unitOfWork)
    {
        _unitOfWork = unitOfWork;
    }

    public async Task TransferInventory(int fromId, int toId, int quantity)
    {
        await _unitOfWork.BeginTransactionAsync();
        try
        {
            // Get products from repository
            var from = await _unitOfWork.Products.GetByIdAsync(fromId);
            var to = await _unitOfWork.Products.GetByIdAsync(toId);

            // Modify inventory
            from.Quantity -= quantity;
            to.Quantity += quantity;

            // Single save for all changes
```

Multiple Database Providers

EF Core supports various database systems with provider-specific packages

SQL SERVER

```
// Package: Microsoft.EntityFrameworkCore.SqlServer
services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer(connectionString));
```

POSTGRESQL

```
// Package: Npgsql.EntityFrameworkCore.PostgreSQL
services.AddDbContext<AppDbContext>(options =>
    options.UseNpgsql(connectionString));
```

SQLITE

```
// Package: Microsoft.EntityFrameworkCore.Sqlite
services.AddDbContext<AppDbContext>(options =>
    options.UseSqlite("Data Source=app.db"));
```

MYSQL

```
// Package: Pomelo.EntityFrameworkCore.MySql
services.AddDbContext<AppDbContext>(options =>
    options.UseMySql(connectionString,
        ServerVersion.AutoDetect(connectionString)));
```

COSMOS DB

```
// Package: Microsoft.EntityFrameworkCore.Cosmos
services.AddDbContext<AppDbContext>(options =>
    options.UseCosmos(endpoint, key, dbName));
```

IN-MEMORY

```
// Package: Microsoft.EntityFrameworkCore.InMemory
services.AddDbContext<AppDbContext>(options =>
    options.UseInMemoryDatabase("TestDb"));
```

> You can even use multiple providers in the same application with separate contexts!

EF Core with In-Memory Database

Perfect for unit testing and rapid prototyping

Setup and Configuration

```
// Install: Microsoft.EntityFrameworkCore.InMemory

public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }
    public DbSet<Product> Products { get; set; }
}

// Create in-memory context
var options = new DbContextOptionsBuilder<AppDbContext>()
    .UseInMemoryDatabase(databaseName: "TestDb")
    .Options;

using var context = new AppDbContext(options);
```

Benefits and Limitations

Benefits:

Unit Testing with EF Core

Writing testable data access code

```
public class ProductServiceTests
{
    private AppDbContext GetInMemoryContext()
    {
        var options = new DbContextOptionsBuilder<AppDbContext>()
            .UseInMemoryDatabase(databaseName: Guid.NewGuid().ToString())
            .Options;
        return new AppDbContext(options);
    }

    [Fact]
    public async Task GetExpensiveProducts_ReturnsCorrectProducts()
    {
        // Arrange
        using var context = GetInMemoryContext();
        context.Products.AddRange(
            new Product { Name = "Cheap", Price = 10 },
            new Product { Name = "Expensive", Price = 200 },
            new Product { Name = "Very Expensive", Price = 500 }
        );
        await context.SaveChangesAsync();
        var service = new ProductService(context);
```

EF Core Joins

Querying related data across multiple tables

INNER JOIN (LINQ QUERY)

```
var query = from product in context.Products
            join category in context.Categories
                on product.CategoryId
                equals category.Id
            select new
            {
                ProductName = product.Name,
                CategoryName = category.Name
            };
```

INNER JOIN (METHOD SYNTAX)

```
var query = context.Products
    .Join(context.Categories,
        p => p.CategoryId,
        c => c.Id,
        (p, c) => new
        {
            ProductName = p.Name,
            CategoryName = c.Name
        });
```

INCLUDE (EAGER LOADING)

```
// Load related entities
var products = await context.Products
    .Include(p => p.Category)
    .Include(p => p.Supplier)
    .ToListAsync();

// Multi-level include
var orders = await context.Orders
    .Include(o => o.Customer)
    .Include(o => o.OrderItems)
        .ThenInclude(oi => oi.Product)
    .ToListAsync();
```

LEFT JOIN

```
var query = from p in context.Products
            join c in context.Categories
                on p.CategoryId equals c.Id
                into categoryGroup
            from c in categoryGroup
                .DefaultIfEmpty()
```

EF Core Aggregation

Computing summary values and statistics

BASIC AGGREGATIONS

```
// Count
var total = await context.Products
    .CountAsync();

// Sum
var totalValue = await context.Products
    .SumAsync(p => p.Price * p.Quantity);

// Average
var avgPrice = await context.Products
    .AverageAsync(p => p.Price);

// Min/Max
var cheapest = await context.Products
    .MinAsync(p => p.Price);
var expensive = await context.Products
    .MaxAsync(p => p.Price);
```

GROUPING

HAVING CLAUSE

```
var popularCategories =
    await context.Products
    .GroupBy(p => p.Category)
    .Where(g => g.Count() > 10)
    .Select(g => new
    {
        Category = g.Key,
        ProductCount = g.Count()
    })
    .ToListAsync();
```

COMPLEX AGGREGATIONS

```
var summary = await context.Orders
    .GroupBy(o => new
    {
        o.CustomerId,
        Year = o.OrderDate.Year
    })
    .Select(g => new
    {
```

Custom SQL Queries

When LINQ isn't sufficient for complex scenarios

FROMSQLRAW

```
var products = await context.Products
    .FromSqlRaw(@"
        SELECT * FROM Products
        WHERE Price > {0}", 100)
    .ToListAsync();

// Composable with LINQ
var filtered = await context.Products
    .FromSqlRaw("SELECT * FROM Products")
    .Where(p => p.InStock)
    .OrderBy(p => p.Name)
    .ToListAsync();
```

FROMSQLINTERPOLATED (SAFER)

```
var minPrice = 100m;
var products = await context.Products
    .FromSqlInterpolated($@"
        SELECT * FROM Products
        WHERE Price > {minPrice}")
```

EXECUTESQLRAW

```
// For UPDATE/DELETE operations
var affected = await context.Database
    .ExecuteSqlRawAsync(@"
        UPDATE Products
        SET Price = Price * 1.1
        WHERE CategoryId = {0}",
        categoryId);
```

STORED PROCEDURES

```
var products = await context.Products
    .FromSqlRaw(
        "EXEC GetProductsByCategory {0}",
        categoryId)
    .ToListAsync();

// With output parameters
var outputParam = new SqlParameter
{
    ParameterName = "@TotalCount",
    SqlDbType = SqlDbType.Int,
```

EF Core Configuration

Configuring DbContext and entity behavior

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

    public DbSet<Product> Products { get; set; }
    public DbSet<Category> Categories { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        if (!optionsBuilder.IsConfigured)
        {
            optionsBuilder
                .UseSqlServer("ConnectionString")
                .EnableSensitiveDataLogging() // ⚠️ Development only!
                .LogTo(Console.WriteLine, LogLevel.Information);
        }
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        // Configure entity
```

Entity Configuration Classes

Separating configuration logic for better organization

CONFIGURATION CLASS

```
public class ProductConfiguration
    : IEntityTypeConfiguration<Product>
{
    public void Configure(
        EntityTypeBuilder<Product> builder)
    {
        builder.ToTable("Products");
        builder.HasKey(p => p.Id);

        builder.Property(p => p.Name)
            .IsRequired()
            .HasMaxLength(200);

        builder.Property(p => p.Price)
            .HasColumnType("decimal(18,2)")
            .IsRequired();

        builder.HasIndex(p => p.Sku)
            .IsUnique();

        builder.HasOne(p => p.Category)
```

APPLYING CONFIGURATION

```
protected override void OnModelCreating(
    ModelBuilder modelBuilder)
{
    // Single configuration
    modelBuilder.ApplyConfiguration(
        new ProductConfiguration());

    // All from assembly
    modelBuilder
        .ApplyConfigurationsFromAssembly(
            typeof(AppDbContext).Assembly);
}
```

BENEFITS

- ✅ Separation of concerns
- ✅ Reusable configurations
- ✅ Cleaner DbContext
- ✅ Easier maintenance

Connection Strings & Configuration

Setting up database connections in ASP.NET Core

APPSETTINGS.JSON

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=localhost;Database=MyApp;Trusted_Connection=true;",
    "PostgreSQL": "Host=localhost;Database=myapp;Username=user;Password=pass"
  }
}
```

PROGRAM.CS CONFIGURATION

```
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");

builder.Services.AddDbContext<AppDbContext>(options =>
{
    options.UseSqlServer(connectionString, sqlOptions =>
    {
        sqlOptions.EnableRetryOnFailure(
            maxRetryCount: 5,
            maxRetryDelay: TimeSpan.FromSeconds(30),
            errorNumbersToAdd: null);

        sqlOptions.CommandTimeout(60);
```

Advanced Configuration Options

DBCONTEXT POOLING

```
// More efficient for web apps
builder.Services.AddDbContextPool
    <AppDbContext>(
    options => options.UseSqlServer(
        connectionString),
    poolSize: 128);
```

CONNECTION RESILIENCY

```
options.UseSqlServer(
    connectionString,
    sqlOptions =>
    {
        sqlOptions.EnableRetryOnFailure(
            maxRetryCount: 5,
            maxRetryDelay:
                TimeSpan.FromSeconds(30),
            errorNumbersToAdd: null
        );
    }
);
```

MULTIPLE CONTEXTS

```
builder.Services.AddDbContext<AppDbContext>(
    options => options.UseSqlServer(
        connectionString1));

builder.Services.AddDbContext
    <IdentityDbContext>(
    options => options.UseSqlServer(
        connectionString2));

builder.Services.AddDbContext
    <LoggingDbContext>(
    options => options.UseNpgsql(
        connectionString3));
```

QUERY TRACKING BEHAVIOR

```
// Global setting
options.UseQueryTrackingBehavior(
    QueryTrackingBehavior.NoTracking);

// Per-query override
var products = context.Products
```

Performance Tips

Optimizing EF Core queries for better performance

USE ASNOTRACKING

```
// For read-only queries
var products = await context.Products
    .AsNoTracking()
    .ToListAsync();
```

PROJECTIONS (SELECT ONLY WHAT YOU NEED)

```
var names = await context.Products
    .Select(p => p.Name)
    .ToListAsync();
```

SPLIT QUERIES

```
// Avoid cartesian explosion
var blogs = await context.Blogs
    .Include(b => b.Posts)
    .Include(b => b.Contributors)
    .AsSplitQuery()
    .ToListAsync();
```

COMPILED QUERIES

BATCH OPERATIONS (EF CORE 7+)

```
await context.Products
    .Where(p => p.Discontinued)
    .ExecuteDeleteAsync();

await context.Products
    .Where(p => p.CategoryId == 1)
    .ExecuteUpdateAsync(
        s => s.SetProperty(
            p => p.Price,
            p => p.Price * 1.1m));
```

AVOID N+1 QUERIES

```
// ❌ Bad: N+1 queries
foreach (var order in orders)
{
    var customer = await context.Customers
        .FindAsync(order.CustomerId);
}

// ✅ Good: Single query
```

Best Practices Summary

Thank You!

Questions?

RESOURCES

- 📚 EF Core Documentation
- 🐙 GitHub Repository
- ⚡ EF Core Performance
- 🎓 Microsoft Learn - EF Core

KEY TAKEAWAYS

- Entity Framework simplifies data access with LINQ
- Repository pattern improves testability
- Multiple database providers for flexibility
- In-memory database perfect for unit testing
- Performance optimization is crucial for production apps