WEEK-5 NOTES

▼ LINQ in C#

LINQ (Language Integrated Query) is a powerful query language introduced in .NET 3.5 that allows developers to query data from different sources (like collections, databases, XML, etc.) in a consistent, type-safe, and readable manner.

LINQ Syntax Types

Query Syntax (SQL-like)

```
var result = from x in collection
             where x.Age > 20
             orderby x.Name
             select x;
```

Method Syntax (Lambda Expressions)

```
var result = collection
             .Where(x => x.Age > 20)
             .OrderBy(x => x.Name)
             .Select(x => x);
```

Both are compiled to the same IL code internally.

LINQ Operations

Filtering (`Where`)

```
var highSalary = employees.Where(e => e.Salary > 60000);
```

Projection (`Select`)

```
var names = employees.Select(e => e.Name);
```

Sorting (`OrderBy`, `OrderByDescending`, `ThenBy`)

```
var sorted = employees
        .OrderBy(e => e.Department)
        .ThenByDescending(e => e.Salary);
```

Grouping

```
var groupByDept = employees
    .GroupBy(e => e.Department)
    .Select(g => new
    {
        Department = g.Key,
        AvgSalary = g.Average(x => x.Salary),
        Employees = g.Count()
    });
```

- `GroupBy(e => e.Department)` → creates groups where all employees having the same department are grouped together.
- Each group is an `IGrouping<TKey, TElement>`.
    - `Key` = department name (e.g., “IT”, “HR”, “Finance”)
    - Group elements = list of employees belonging to that key.
- Then `Select()` projects each group into an anonymous object:
    - `.Key` → department name
    - `.Average()` → average salary of that department
    - `.Count()` → number of employees in that group

Aggregation (`Count`, `Sum`, `Average`, `Max`, `Min`)

```
int totalEmp = employees.Count();
double avgSalary = employees.Average(e => e.Salary);
```

Join

```
var result = employees.Join(
departments,                  // inner collection
e => e.Department,            // outer key selector
d => d.DeptName,              // inner key selector
(e, d) => new { e.Name, d.DeptName } // result selector
);
```

- `.Join()` matches each `employee.Department` with corresponding `department.DeptName`
- If keys match → creates a new anonymous object containing both.
- This works like **INNER JOIN** in SQL.

Set Operations (`Distinct`, `Union`, `Intersect`, `Except`)

```
var dept1 = new[] { "IT", "Finance", "HR" };
var dept2 = new[] { "Finance", "Admin" };

var union = dept1.Union(dept2);      // all unique
var common = dept1.Intersect(dept2);  // common
var diff = dept1.Except(dept2);      // in dept1 not in dept2
```

Quantifier Operators (`Any`, `All`, `Contains`)

```
bool hasHigh = employees.Any(e => e.Salary > 70000);
bool allAbove20 = employees.All(e => e.Age > 20);
```

Skip and Take (Pagination)

```
var page2 = employees.Skip(2).Take(2);
```

Element Operators (`First` , `FirstOrDefault` , `Single` , `ElementAt`)

```
var firstEmp = employees.First();
var thirdEmp = employees.ElementAt(2);
var notFound = employees.FirstOrDefault(e => e.Name == "XYZ");
```

- `First()` throws exception if empty.
- `FirstOrDefault()` returns `null` .

Conversion Operators (`ToList` , `ToArray` , `ToDictionary`)

```
var empDict = employees.ToDictionary(e => e.Id, e => e.Name);
```

LINQ Loops Concept

LINQ queries do **not execute immediately** when written.

They execute **only when enumerated** using:

- `foreach`
- `.ToList()`
- `.Count()`
- `.First()` , etc.

Example:

```
var q = employees.Where(e => e.Salary > 50000); // not  executed

employees.Add(new Employee { Name = "Jay", Department = "IT", Salar
y = 80000 });

foreach (var e in q)  // executed now
    Console.WriteLine(e.Name);
```

LINQ with DataTable

Filtering and projection

```
var result = dt.AsEnumerable()
.Where(r => r.Field<double>("Salary") > 60000)
.Select(r => new
{
Name = r.Field<string>("Name"),
Dept = r.Field<string>("Department"),
Salary = r.Field<double>("Salary")
});
```

Grouping and projection

```
var groupByDept = dt.AsEnumerable()
.GroupBy(r => r.Field<string>("Department"))
.Select(g => new
{
Department = g.Key,
AvgSalary = g.Average(x => x.Field<double>("Salary")),
Count = g.Count()
});
```

Parallel LINQ (PLINQ)

To utilize multiple cores:

```
var result = employees.AsParallel()
    .Where(e => e.Salary > 50000)
    .Select(e => e.Name);
```

▼ Dynamic Type

The `dynamic` type in C# was introduced in **C# 4.0** (.NET 4.0).

It tells the compiler to **defer type checking until runtime.**

```
dynamic value = "Hello, World!";
Console.WriteLine(value.Length);  // Works (string has Length)
value = 123;
Console.WriteLine(value + 10);    // Works (int + int)
```

Here:

- The **compiler** doesn't verify if `value` has `.Length`.
- The **runtime** checks it dynamically.

`dynamic` vs `var` vs `object`

Feature	`dynamic`	`var`	`object`
Type checking	At **runtime**	At **compile-time**	At **compile-time**
Type inference	Runtime dynamic	Compile-time inferred	Static
Members accessible	Any — compiler trusts you	Only those of inferred type	Only those of `object` (need casting)
Performance	Slower (runtime binder)	Fast (statically compiled)	Moderate
Usage	Interop, Reflection, dynamic JSON/XML	Normal code	Generic containersHow It Works Internally

When you declare a variable as `dynamic`, it's actually stored as `object`,

but all operations on it are handled via the **Dynamic Language Runtime (DLR)**.

DLR (Dynamic Language Runtime):

- Sits on top of the CLR (Common Language Runtime).
- Provides runtime binding for dynamic operations.

```
dynamic expando = new ExpandoObject();
expando.Name = "Nisharg";
```

```
Console.WriteLine(expando.Name);
```

```
object obj = new SomeClass();
var method = obj.GetType().GetMethod("Test");
method.Invoke(obj, null);
```

You can do:

```
dynamic obj = new SomeClass();
obj.Test();
```

Dynamic JSON / XML Data

When deserializing JSON without defining a strict class:

```
string json = "{ \"Name\": \"Nisharg\", \"Age\": 21 }";
dynamic data = JsonConvert.DeserializeObject<dynamic>(json);
Console.WriteLine(data.Name);  // Nisharg
```

Or XML:

```
dynamic xml = new XmlDocument();
xml.LoadXml("<Person><Name>Nisharg</Name></Person>");
Console.WriteLine(xml.Person.Name.InnerText);
```

Corner Cases

Runtime Errors Instead of Compile-Time

```
dynamic x = "Hello";
Console.WriteLine(x.Lenght);  // Typo! RuntimeBinderException
```

Compiler won’t warn you — fails at runtime.

Performance Overhead

Every dynamic call goes through:

- DLR binding
- Reflection metadata lookup
- Runtime caching

▼ Cryptography

Cryptography means protecting information by transforming (encrypting) it so only authorized people can read or modify it.

Common Goals of Cryptography:

Goal	Meaning	Example
Confidentiality	Keep data secret	Encrypt a password before saving
Integrity	Detect if data is changed	Hash a file and compare later
Authentication	Confirm identity	Verify a digital signature
Non-repudiation	Cannot deny an action	Signed emails / documents

C# Cryptography Library Overview

```
using System.Security.Cryptography;
```

The important classes:

Category	Common Class	Purpose
Symmetric Encryption	`Aes`, `AesGcm`, `AesCng`	Same key for encryption and decryption
Asymmetric Encryption	`RSA`, `ECDsa`	Public/Private key-based encryption
Hashing	`SHA256`, `SHA512`, `MD5`	One-way data fingerprints
HMAC	`HMACSHA256`	Integrity check with a secret key
Key Derivation	`Rfc2898DeriveBytes`	Generate strong keys from passwords

Category	Common Class	Purpose
Random Numbers	`RandomNumberGenerator`	Generate secure random keys
Digital Signature	`RSA`, `ECDsa`	Sign & verify data

Symmetric Encryption (AES)

What is AES?

AES (Advanced Encryption Standard) uses a *secret key* to encrypt and decrypt data.

Same key = symmetric encryption.

C# provides classes:

- `Aes` (basic AES)
- `AesGcm` (modern AES with authentication)
- `AesCbc` (older mode, needs IV)

AES-GCM (Modern & Secure)

AES-GCM provides **authenticated encryption**, meaning it both encrypts and protects from tampering.

```
 using System;
using System.Security.Cryptography;

class AESGcmDemo
{
    static void Main()
    {
        byte[] key = new byte[32]; // 256-bit key
        RandomNumberGenerator.Fill(key);

        byte[] nonce = new byte[12]; // 96-bit nonce
        RandomNumberGenerator.Fill(nonce);

        string text = "Confidential data!";
```

```
        byte[] plaintext = System.Text.Encoding.UTF8.GetBytes(text);
        byte[] ciphertext = new byte[plaintext.Length];
        byte[] tag = new byte[16]; // authentication tag

        using (AesGcm aes = new AesGcm(key))
        {
            aes.Encrypt(nonce, plaintext, ciphertext, tag);
        }

        Console.WriteLine("Encrypted: " + Convert.ToBase64String(ciphert
ext));

        // Decrypt
        byte[] decrypted = new byte[ciphertext.Length];
        using (AesGcm aes = new AesGcm(key))
        {
            aes.Decrypt(nonce, ciphertext, tag, decrypted);
        }

        Console.WriteLine("Decrypted: " + System.Text.Encoding.UTF8.Get
String(decrypted));
    }
}
```

Hashing (SHA256)

Hashing converts data into a fixed-size unique value — like a fingerprint.

Used for:

- Storing passwords (with salt)
- Checking integrity of files

Example:

```
public static string SHA256Hash(string input)
{
    using (SHA256 sha256 = SHA256.Create())
```

```
    {
        byte[] bytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(in
put));

        return Convert.ToHexString(bytes);
    }
  }
  static void Main(string[] args)
  {
    string input = "Nisharg";
    string hash = SHA256Hash(input);
    Console.WriteLine($"Input: {input}");
    Console.WriteLine($"SHA-256 Hash: {hash}");
  }
```

Keyed Hash (HMAC)

HMAC = Hash + Secret key.

Used to check integrity (to detect tampering).

```
using System;
using System.Security.Cryptography;
using System.Text;

class HmacDemo
{
    static void Main()
    {
        byte[] key = Encoding.UTF8.GetBytes("secretkey");
        string message = "Important Data";

        using var hmac = new HMACSHA256(key);
        byte[] tag = hmac.ComputeHash(Encoding.UTF8.GetBytes(messag
e));

        Console.WriteLine("HMAC: " + Convert.ToBase64String(tag));
```

```
    }
}
```

Key Derivation (PBKDF2)

To turn a password into a strong encryption key:

Use `Rfc2898DeriveBytes` (PBKDF2).

```
using System;
using System.Security.Cryptography;
using System.Text;

class Pbkdf2Demo
{
    static void Main()
    {
        string password = "mypassword";
        byte[] salt = new byte[16];
        RandomNumberGenerator.Fill(salt);

        int iterations = 100000;
        using var pbkdf2 = new Rfc2898DeriveBytes(password, salt, iterati
ons, HashAlgorithmName.SHA256);
        byte[] key = pbkdf2.GetBytes(32); // 256-bit key

        Console.WriteLine("Derived Key: " + Convert.ToBase64String(ke
y));
    }
}
```

Why use PBKDF2?

Because if someone steals your database, they can’t easily brute-force the password.

Asymmetric Encryption (RSA)

RSA uses two keys:

- **Public key** (for encryption or verifying signature)
- **Private key** (for decryption or signing)

Example: RSA Encrypt & Decrypt

```
using System;
using System.Security.Cryptography;
using System.Text;

class RsaDemo
{
    static void Main()
    {
        using var rsa = RSA.Create();

        // Export keys
        RSAParameters pub = rsa.ExportParameters(false);
        RSAParameters priv = rsa.ExportParameters(true);

        byte[] data = Encoding.UTF8.GetBytes("Secret message");

        byte[] encrypted = rsa.Encrypt(data, RSAEncryptionPadding.OaepSHA256);
        Console.WriteLine("Encrypted: " + Convert.ToBase64String(encrypted));

        byte[] decrypted = rsa.Decrypt(encrypted, RSAEncryptionPadding.OaepSHA256);
        Console.WriteLine("Decrypted: " + Encoding.UTF8.GetString(decrypted));
    }
}
```

Digital Signatures

Use digital signatures to ensure that the message is from the sender and not modified.

Example: RSA Signature

```
using System;
using System.Security.Cryptography;
using System.Text;

class SignDemo
{
    static void Main()
    {
        using var rsa = RSA.Create();
        byte[] data = Encoding.UTF8.GetBytes("Authenticate this messag
e");

        byte[] signature = rsa.SignData(data, HashAlgorithmName.SHA256,
RSASignaturePadding.Pss);
        Console.WriteLine("Signature: " + Convert.ToBase64String(signatu
re));

        bool verified = rsa.VerifyData(data, signature, HashAlgorithmName.
SHA256, RSASignaturePadding.Pss);
        Console.WriteLine("Verified: " + verified);
    }
}
```

Random Numbers

Never use `Random`.

Use `RandomNumberGenerator` for cryptographic purposes.

```
byte[] key = new byte[32];
RandomNumberGenerator.Fill(key);
```

▼ ORM

ORM stands for **Object-Relational Mapping** .

It's a technique that allows developers to interact with a **relational database** (like SQL Server, MySQL, PostgreSQL) using **object-oriented programming (OOP)** concepts instead of writing raw SQL queries.

How ORM Works (in C#)

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

Students table in database.

Id	Name	Age
1	Raj	20
2	Neha	22

The ORM maps:

- `Student` class → `Students` table
- `Id` , `Name` , `Age` properties → columns `Id` , `Name` , `Age`

instead of writing SQL like:

```
SELECT * FROM Students;
```

just write in C#

```
var students = context.Students.ToList();
```

The ORM tool automatically generates SQL behind the scenes and returns results as a **list of Student objects**.

Popular ORM Tools in C#

1. **Entity Framework (EF Core)**
2. **Dapper**
3. **NHibernate**

EF Core

Packages

For SQL Server:

```
dotnet add package Microsoft.EntityFrameworkCore
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet add package Microsoft.EntityFrameworkCore.Design
```

For SQLite:

```
dotnet add package Microsoft.EntityFrameworkCore.Sqlite
```

For MySQL

```
dotnet add package Pomelo.EntityFrameworkCore.MySql
or
dotnet add package MySql.EntityFrameworkCore
```

Fundamental Concepts

- **Entity**: A CLR class mapped to a table.
- **DbContext**: The unit-of-work / repository-ish object that represents a session with the database. Holds `DbSet<TEntity>` properties.
- **DbSet<TEntity>**: Represents a tabl queryable through LINQ.
- **Model**: The set of entity types plus relationships and mapping metadata. Built from conventions, data annotations, and Fluent API.

- **Migrations**: Track schema changes and apply them to DB.
- **Change Tracker**: Tracks entity states (Added, Modified, Deleted, Unchanged, Detached).
- **Query Types / Keyless Entities**: For views, raw SQL results, or DTO-like projections.

Creating Models

Make One folder Data in that make files for each and every table as differnet classes

```
public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Description { get; set; }
    public string Author { get; set; }

}
```

For different Types of relationships

1. One-to-One (1:1) Relationship

```
public class User
{
    public int Id { get; set; }
    public string Name { get; set; }
    public Profile Profile { get; set; }  // Navigation Property
}

public class Profile
{
    public int Id { get; set; }
    public string Bio { get; set; }
```

```
    public int UserId { get; set; }  // Foreign Key
    public User User { get; set; }   // Navigation Property
}
```

2. One-to-Many (1:M) Relationship
Example: A **Category** has **many** Products.

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

    public virtual ICollection<Product> Products { get; set; } = new List<Product>(); // Navigation Property
}

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

    public int CategoryId { get; set; } // Foreign Key
    public Category Category { get; set; } // Navigation Property
}
```

3. Many-to-Many (M:N) Relationship

Example: A **Student** can enroll in **many Courses**, and a **Course** can have **many Students.**

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

    public ICollection<Course> Courses { get; set; } = new List<Course>(); // Navigation Property
}
```

```
public class Course
{
    public int Id { get; set; }
    public string Title { get; set; }

    public ICollection<Student> Students { get; set; } = new List<Student
>(); // Navigation Property
}
```

No, just writing the navigation property is not always enough!

Entity Framework **does infer relationships** from navigation properties, but **sometimes, explicit configuration is needed**—especially for complex cases.

How EF Determines Relationships

When EF Automatically Sets Relationships

If you follow **conventions** (naming rules), EF **automatically** understands the relationship.

For example:

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

public int CategoryId { get; set; }  // Foreign Key
public Category Category { get; set; }  // Navigation Property
}

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

```
public ICollection<Product> Products { get; set; }  // Navigation Propert
y
}
```

Since `CategoryId` **exists**, EF **automatically** creates a **one-to-many relationship.**

When Explicit Configuration is Required

If you **don't** follow conventions, you must **manually configure relationships** using **Fluent API.**

Example: Configuring One-to-Many Explicitly

```
using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext : DbContext
{
public DbSet<Category> Categories { get; set; }
public DbSet<Product> Products { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
   modelBuilder.Entity<Product>()
      .HasOne(p => p.Category)  // One Product has one Category
      .WithMany(c => c.Products)  // One Category has many Products
      .HasForeignKey(p => p.CategoryId);  // Foreign Key
}
}
```

Use Fluent API when:

- The foreign key is missing.
- The relationship is complex (e.g., many-to-many).
- You need specific behaviors (e.g., cascading delete).

Creating DbContext File

```
public class AppDbContext : DbContext
{
    public DbSet<Category> Categories { get; set; }
    public DbSet<Product> Products { get; set; }

    public AppDbContext(DbContextOptions<AppDbContext> options): b
ase(options) { }

}
```

`Data Seeding`

```
public class AppDbContext : DbContext
{
    public DbSet<Category> Categories { get; set; }
    public DbSet<Product> Products { get; set; }

    public AppDbContext(DbContextOptions<AppDbContext> options): b
ase(options) { }
    protected override void OnModelCreating(ModelBuilder modelBuilde
r)
    {
        modelBuilder.Entity<Category>().HasData(
            new Category() { Id = 1, Name = "Electronics" },
            new Category() { Id = 2, Name = "Clothes" }
            );
        }
    }

}



//or like this

if (!context.Categories.Any()) {
    context.Categories.Add(new Category { Name = "General" });
```

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

Data annotations

```
public class User
{
        [Key] // System.ComponentModel.DataAnnotations
        public int UserId { get; set; }
        [Required]
        [MaxLength(100)]
        public string Username { get; set; }
}
```

```
Fluent API examples:
modelBuilder.Entity<User>()
.Property(u => u.Username)
.IsRequired()
.HasMaxLength(100);

modelBuilder.Entity<User>()
.HasIndex(u => u.Username)
.IsUnique();

modelBuilder.Entity<Order>()
.Property(o => o.Total)
.HasColumnType("decimal(18,2)");
```

EF Core Migrations

What is a Migration?

A **migration** in EF Core is a **way to evolve your database schema** over time while keeping it in sync with your C# entity classes.

- EF Core tracks changes in your **model (C# classes).**

- Generates **SQL scripts** to apply these changes to the database.
- Ensures **version control** of your database schema.
- Supports **rollback**, **update**, and **seed data**.

Why use migrations?

- Avoid manual changes to DB schema.
- Track changes over time (like Git for DB).
- Make deploying updates safer and repeatable.

How Migrations Work – Behind the Scenes

When you create a migration:

1. EF Core compares **current model** with **last applied migration** (or empty DB if first migration).
2. Generates **C# files** in a **Migrations folder**.
3. Each migration has:
   - **Up() method** → Defines changes to **apply to DB**
   - **Down() method** → Defines how to **rollback** changes

Example:

```
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Employees",
            columns: table => new
            {
                Id = table.Column<int>(nullable: false)
                         .Annotation("MySql:ValueGenerationStrategy", MySqlVal
ueGenerationStrategy.IdentityColumn),
                Name = table.Column<string>(maxLength: 50, nullable: false),
                Department = table.Column<string>(nullable: true),
                Salary = table.Column<decimal>(nullable: false)
```

```
        },
        constraints: table => 
        {
            table.PrimaryKey("PK_Employees", x => x.Id);
        });
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(name: "Employees");
    }
}
```

- **Up()** → Applies changes (creates table here)
- **Down()** → Reverts changes (drops table here)

Folder Structure of Migrations

After creating migrations, EF Core generates a folder (usually called **Migrations**) with files:

```
Migrations/
├─ 202510171350_InitialCreate.cs      // Migration code
├─ 202510171350_InitialCreate.Designer.cs  // Designer metadata
├─ AppDbContextModelSnapshot.cs       // Snapshot of current DB mod
el
```

Explanation:

1. **Migration file** (`.cs`)
    - Contains `Up()` and `Down()` methods
    - Human-readable code for schema changes
2. **Designer file**
    - Auto-generated metadata
    - EF Core uses it internally, **don't edit manually**

3. **Model Snapshot**

    - Represents **current state of your C# model**
    - EF Core compares it with future models to detect changes

Common Migration Commands

Step 1: Add Migration

```
dotnet ef migrations add MigrationName
```

- Generates a new migration file in `Migrations/`
- Example:

```
dotnet ef migrations add AddDepartmentTable
```

Step 2: Apply Migration

```
dotnet ef database update
```

- Applies **all pending migrations** to DB

Step 3: Apply Specific Migration

```
dotnet ef database update InitialCreate
```

- Rolls DB forward/backward to a specific migration

Step 4: Remove Last Migration

```
dotnet ef migrations remove
```

- Deletes **last migration** (only if not applied to DB)
- Safe way to undo a migration you haven’t applied yet

Step 5: Rollback / Revert Migration

```
dotnet ef database update PreviousMigration
```

- Moves DB **back to a previous migration**
- EF Core runs **Down() methods** to revert changes

Lazy Loading and Eager Loading

Use Package

dotnet add package Microsoft.EntityFrameworkCore.Proxies

Define Models with Virtual Navigation Property

```
public class Product
{
public int Id { get; set; }
public string Name { get; set; }
public decimal Price { get; set; }

// Lazy Loading: virtual keyword enables dynamic proxy loading
public virtual ICollection<Review> Reviews { get; set; } = new List<Revie
w>();
}

public class Review
{
public int Id { get; set; }
public string Comment { get; set; }
public int ProductId { get; set; }
public virtual Product Product { get; set; }
}
```

Enable Lazy Loading in `AppDbContext`

```
public class AppDbContext : DbContext
{
public DbSet<Product> Products { get; set; }
public DbSet<Review> Reviews { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder.UseSqlite("Data Source=products.db")
                  .UseLazyLoadingProxies(); // Enable Lazy Loading
}
}
```

Lazy Loading in Action

```
var product = await db.Products.FindAsync(id);
if (product == null) return Results.NotFound();

var reviews = product.Reviews; // lazy Loading happens here
```

Reviews are loaded only when accessed (`product.Reviews`).
If you iterate over `Reviews`, a separate query is sent to the database each time (N+1 issue).

Eager Loading (Load Everything Upfront)

- EF **loads related data in a single query** using `.Include()` .
- No additional queries when accessing navigation properties.
- **Better for performance** if you always need related data.

Load Reviews with Product in One Query

```
var product = await db.Products
```

```
.Include(p => p.Reviews) // eager Loading
.FirstOrDefaultAsync(p => p.Id == id);
```

Transactions

What is a Transaction?

A **transaction** in EF Core is a way to ensure that multiple database operations either **succeed together or fail together**. If one operation fails, the **entire transaction is rolled back**, preventing partial updates that could lead to **inconsistent data**.

Types of Transactions in EF Core

EF Core supports transactions in **three ways**:

1. **Implicit Transactions** (Default Behavior)
2. **Explicit Transactions** (Using `BeginTransaction()`)

Implicit Transactions (Default Behavior)

By default, EF Core wraps `SaveChanges()` inside a transaction.
If `SaveChanges()` fails, **all changes are rolled back**.

Example: Default Transaction in EF Core

```
db.Products.Add(product);
await db.SaveChangesAsync(); // Automatically wrapped in a transaction.
```

If `SaveChangesAsync()` fails, the transaction is automatically rolled back.
Cannot group multiple operations into one transaction.

Explicit Transactions using `BeginTransaction()`

Used when **multiple operations** should be treated as a **single atomic unit.**
If **one operation fails**, the entire transaction is **rolled back.**

Example: Multiple Operations in a Transaction

```
using var transaction = await db.Database.BeginTransactionAsync(); // Begin transaction

try
{
    var product1 = new Product { Name = "Laptop", Price = 1000 };
    var product2 = new Product { Name = "Mouse", Price = 50 };

    db.Products.Add(product1);
    await db.SaveChangesAsync(); // First operation

    db.Products.Add(product2);
    await db.SaveChangesAsync(); // Second operation

    await transaction.CommitAsync(); // Commit the transaction
    return Results.Ok("Products added successfully!");
}
catch (Exception)
{
    await transaction.RollbackAsync(); //  Rollback if any operation fails
    return Results.Problem("Transaction failed, no products added.");
}
```

Soft Delete

Soft delete is a technique where records **are not physically deleted** from the database but instead **marked as deleted** using a flag (e.g., `IsDeleted` column). This ensures that data remains available for auditing or recovery purposes.

```
public class Product
{
public int Id { get; set; }
public string Name { get; set; }
public decimal Price { get; set; }
public bool IsDeleted { get; set; }  //  Soft delete flag
}
```

```
public class AppDbContext : DbContext
{
    public DbSet<Product> Products { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuild
er)
    {
        modelBuilder.Entity<Product>().HasQueryFilter(p => !p.IsDelet
ed); //global filter
    }
}
```

```
var product = await db.Products.IgnoreQueryFilters().FirstOrDefaultAsy
nc(p => p.Id == id);
if (product is null) return Results.NotFound();
product.IsDeleted = false; // Restore product
await db.SaveChangesAsync();
```

▼ Dapper

Dapper is a **micro ORM (Object Relational Mapper)** built by StackOverflow.

It extends `IDbConnection` (like `SqlConnection`, `MySqlConnection`, etc.) to make database work simpler and faster.

So Dapper doesn't have hundreds of classes like EF Core — it's small, fast, and uses **extension methods** on your connection object.

Core Dapper Namespaces & Types

Class / Interface	Description
`SqlMapper`	The **main static class** where all extension methods (like `Query`, `Execute`, etc.) are defined. You rarely call it directly.
`DynamicParameters`	A class for building **complex parameter lists** (input/output, stored procedures).
`GridReader`	Used internally by `QueryMultiple()` to read multiple result sets. You can call `Read<T>()` on it.

Extension Methods on `IDbConnection`

These are the **heart of Dapper** — methods you call directly on your `connection` object.

Data Retrieval (SELECT)

Method	Returns	Description
`Query<T>(sql, params, ...)`	`IEnumerable<T>`	Runs a SQL SELECT and maps each row to `T`.
`Query(sql, params, ...)`	`IEnumerable<dynamic>`	Returns rows as dynamic objects (no model class needed).
`QueryFirst<T>()`	`T`	Returns **first** row; throws if no rows.
`QueryFirstOrDefault<T>()`	`T` or `default`	Returns first row or null if none.
`QuerySingle<T>()`	`T`	Expects **exactly one** row; throws if more or less.
`QuerySingleOrDefault<T>()`	`T` or `default`	Expects one or none.
`QueryMultiple()`	`GridReader`	Executes multiple queries and lets you read them separately.

Example:

```
var result = connection.Query<Person>("SELECT * FROM People");
var one = connection.QuerySingleOrDefault<Person>("SELECT * FROM People WHERE Id = 1");
```

Data Modification (INSERT, UPDATE, DELETE)

Method	Returns	Description
`Execute()`	`int`	Executes non-query SQL; returns affected rows count.
`ExecuteAsync()`	`Task<int>`	Async version.

Example:

```
int rows = connection.Execute("UPDATE Department SET Name=@n WHERE Id=@i", new { n = "HR", i = 1 });
```

Scalar Values

Method	Returns	Description
`ExecuteScalar<T>()`	`T`	Executes a SQL command and returns a **single value**.

Example:

```
int count = connection.ExecuteScalar<int>("SELECT COUNT(*) FROM Employee");
```

Async Versions

All major methods have async equivalents:

Sync	Async Equivalent
`Query<T>()`	`QueryAsync<T>()`
`Execute()`	`ExecuteAsync()`
`ExecuteScalar<T>()`	`ExecuteScalarAsync<T>()`
`QueryMultiple()`	`QueryMultipleAsync()`

Example:

```
var employees = await connection.QueryAsync<Employee>("SELECT * FROM Employee");
```

Multi-Mapping (for JOINs)

Method	Description
`Query<TFirst, TSecond, TResult>()`	Maps joined data into multiple objects.
`Query<TFirst, TSecond, TThird, TResult>()`	For 3 joined tables.

`DynamicParameters` Class

Used for:

- Adding parameters dynamically.
- Handling input/output parameters.
- Stored procedure parameters.

Method	Description
`Add(name, value, dbType, direction, size)`	Add parameter.
`Get<T>(name)`	Get value (for output).
`Remove(name)`	Remove parameter.
`ParameterNames`	Returns all parameter names.

Example:

```
var param = new DynamicParameters();
param.Add("Id", 1);
param.Add("Count", dbType: DbType.Int32, direction: ParameterDirection.Output);

connection.Execute("sp_CountEmployees", param, commandType: CommandType.StoredProcedure);

int count = param.Get<int>("Count");
```

▼ Ormlite

Connection Management

These are the starting points for all database interaction.

- `OrmLiteConnectionFactory`: The class you instantiate once at startup. It holds your connection string and the database dialect.
- `.Open()` / `.OpenDbConnection()`: The primary method on your factory to get an `IDbConnection`. Always wrap this in a `using` block.
- `.OpenAsync()` / `.OpenDbConnectionAsync()`: The asynchronous version to get a database connection.

Basic CRUD (Create, Read, Update, Delete)

These are your fundamental data operations. (e.g., `T` is your POCO, like `User`).

- `db.Insert<T>(obj, ...)`: Inserts a new POCO into the database. Can also take multiple objects.
- `db.InsertAsync<T>(obj, ...)`: Asynchronous version of `Insert`.
- `db.Save<T>(obj, ...)`: A smart "upsert." If the primary key has a value, it calls `Update`. If not, it calls `Insert`.
- `db.SaveAsync<T>(obj, ...)`: Asynchronous version of `Save`.
- `db.Update<T>(obj, ...)`: Updates an existing POCO in the database (based on its Primary Key).
- `db.UpdateAsync<T>(obj, ...)`: Asynchronous version of `Update`.
- `db.Delete<T>(obj, ...)`: Deletes a row based on the object's Primary Key.
- `db.DeleteById<T>(id)`: Deletes a row directly by its Primary Key value.
- `db.DeleteAsync<T>(...)` / `db.DeleteByIdAsync<T>(...)`: Asynchronous versions of `Delete` methods.

- `db.GetLastInsertId()`: A utility to get the last auto-incremented ID from the connection.

Querying & Selecting Data (The `Select` APIs)

This is the largest and most-used part of OrmLite.

Simple Lookups

- `db.Select<T>()`: Selects all rows from the `T` table (`SELECT * FROM ...`).
- `db.SingleById<T>(id)`: Fetches a single record by its Primary Key. Throws an error if not found.
- `db.Single<T>(expression)`: Fetches a single record matching a LINQ expression (e.g., `db.Single<User>(x => x.Email == "a@b.com")`). Throws an error if not found or if more than one is found.
- `db.Exists<T>(expression)`: Returns `true` or `false` if a record matching the criteria exists.
- `db.Count<T>()`: Returns the total count of rows in the table.

Advanced Querying (Typed `SqlExpression`)

This is the fluent query builder.

- `db.From<T>()`: **The starting point.** This returns an `ISqlExpression<T>` object that you build upon.
- `.Where(expression)`: Adds a `WHERE` clause (e.g., `.Where(x => x.Age > 25)`).
- `.And(expression)` / `.Or(expression)`: Appends conditions to your `WHERE` clause.
- `.Join<T2>(...)` / `.LeftJoin<T3>(...)`: Adds joins to other tables.
- `.OrderBy(expression)` / `.OrderByDescending(expression)`: Adds an `ORDER BY` clause.
- `.Skip(rows)`: Adds an `OFFSET` for paging.
- `.Take(rows)`: Adds a `LIMIT` or `TOP` for paging.
- `.GroupBy(expression)`: Adds a `GROUP BY` clause.
- `.Having(expression)`: Adds a `HAVING` clause (used with `GroupBy`).

- `.Select(expression)`: Projects the results into a custom shape (e.g., an anonymous type or a custom DTO).
- `db.Select(expression)`: **The execution method.** You pass the `ISqlExpression` you built (from `db.From<T>()`) to this method to get the results.
- `db.Single(expression)`: Executes the `ISqlExpression` and returns a single record.

Scalar & Column Lookups

- `db.Scalar<T>(expression)`: Executes a query and returns a single value (scalar) of type `T` (e.g., `int`, `string`, `DateTime`).
  - `var count = db.Scalar<int>(db.From<User>().Select(Sql.Count("*")));`
- `db.Column<T>(expression)`: Selects only the first column from the query results and returns them as a `List<T>`.
- `db.ColumnDistinct<T>(expression)`: Same as `Column`, but only returns unique (`DISTINCT`) values.

Raw SQL & Stored Procedures

For when you need complete control.

- `db.SqlList<T>(sql, params)`: Executes any raw SQL `SELECT` query and maps the results to a `List<T>`. Use an anonymous object for parameters.
  - `db.SqlList<User>("SELECT * FROM User WHERE Age > @age", new { age = 30 });`
- `db.SqlScalar<T>(sql, params)`: Executes raw SQL and returns a single scalar value.
  - `db.SqlScalar<int>("SELECT COUNT(*) FROM User");`
- `db.ExecuteSql(sql, params)`: Executes any raw SQL that does not return results (e.g., `UPDATE`, `DELETE`, `INSERT`, `ALTER TABLE`).
- `db.SqlProc(procName, params)`: **For stored procedures.** This returns an `IDbCommand` object.
  - `cmd.AddParam(name, value, direction)`: Used with `SqlProc` to add `ParameterDirection.Output` parameters.

  - `cmd.ConvertToList<T>()` : Used after `SqlProc` to get the results from the command.
  - `cmd.ExecNonQuery()` : Used after `SqlProc` if the procedure doesn't return a result set.

Transaction Management

For controlling units of work.

- `db.OpenTransaction()` : Starts a new database transaction. Returns an `IDbTransaction` that should be in a `using` block.
- `trans.Commit()` : Commits the transaction if all operations succeeded.
- `trans.Rollback()` : Rolls back the transaction if any operation failed.

Schema / DDL (Data Definition Language)

For creating and modifying your database tables (mostly for development/testing).

- `db.CreateTable<T>(overwrite)` : Creates a table based on the POCO `T`.
- `db.CreateTableIfNotExists<T>()` : The safest way to create a table.
- `db.DropTable<T>()` : Drops the table for POCO `T`.
- `db.TableExists<T>()` : Returns `true` if the table exists.
- `db.AlterTable<T>(...)` : A group of methods to modify an existing table.
- `db.AddColumn<T>(expression)` : Adds a new column to a table.
- `db.DropColumn<T>(columnName)` : Drops a column from a table.
- `db.AlterColumn<T>(...)` : Modifies an existing column's type.
- `db.RenameColumn<T>(...)` : Renames a column.