✅ 1. Entity Framework (EF / EF Core) Overview

Entity Framework (EF Core) is an **ORM (Object-Relational Mapper)** for .NET.
It helps developers work with a database **using C# classes** instead of writing SQL queries manually.

🔍 Without EF Core

You must write SQL queries:

```
SELECT * FROM Products WHERE Id = 1;
```

🔍 With EF Core

You write C# code:

```
var product = context.Products.Find(1);
```

EF Core internally converts this into SQL and executes it on the database.

✔ Benefits of using EF Core

Benefit	**Explanation**
Productivity	Less SQL → faster development
Maintainability	Cleaner code; models represent tables
Database portability	Change SQL Server → PostgreSQL → SQLite with minimal code changes
Abstraction	EF handles connection, command creation, data reading
Change tracking	Automatically detects modified objects

Analogy:
Think of EF Core as a **translator**.
You speak C# → EF translates it into SQL → DB responds → EF translates back into C# objects.

✅ 1.1 Introduction to ORM

ORM = **Object Relational Mapping**

👉 Why do we need ORM?

Databases store data in **tables** (rows, columns).
Applications work with **objects** (classes, properties).

ORM bridges this gap.

ORM Responsibilities

- Map **class → table**
- Map **property → column**
- Execute SQL automatically
- Track changes
- Handle relationships (1–1, 1–many, many–many)

Example Mapping

C# Class Property	DB Column
Product.Id	Products.Id
Product.Name	Products.Name
Product.Price	Products.Price

ORM automatically keeps both worlds in sync.

✅ 2. Data Providers in EF Core

Data providers allow EF Core to work with different database engines.

Common EF Core Providers

Provider	Package
SQL Server	Microsoft.EntityFrameworkCore.SqlServer
SQLite	Microsoft.EntityFrameworkCore.Sqlite
PostgreSQL	Npgsql.EntityFrameworkCore.PostgreSQL
MySQL	Pomelo.EntityFrameworkCore.MySql

Each provider knows:

- SQL syntax of that database
- Features supported (identity columns, sequences, etc.)
- Datatypes mapping

Example: Register SQL Server Provider

```
services.AddDbContext<AppDbContext>(options =>
   options.UseSqlServer("connection-string"));
```

Switch to PostgreSQL:

```
options.UseNpgsql("connection-string");
```

👉 **Only change is the provider — EF Core code stays same.**

✅ 3. Programming Models: Code First vs DB First

Entity Framework supports two main workflows.

⭐ 3.1 Code First Approach

You **start with C# classes**, EF Core creates the database.

Steps:

1. Create POCO classes (Product, Order, etc.)
2. Configure DbContext
3. Run migrations → EF generates tables

Example Model

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

When to use Code First?

✓ When starting a new project
✓ When you want EF Core to manage DB structure
✓ Agile teams → evolve model frequently

⭐ 3.2 Database First Approach

You already have a **database**, EF generates:

- C# entity classes
- DbContext with mappings

Command Example:

```
Scaffold-DbContext "connection-string" Microsoft.EntityFrameworkCore.SqlServer
```

When to use DB First?

✓ Working with legacy/enterprise databases
✓ Database designed by DBAs
✓ Complex stored procedures exist

✅ 4. DbContext and DbSet

⭐ 4.1 DbContext

DbContext = **the main class that represents a session with the database**.

It manages:

- Connection to DB
- Change tracking
- Query execution
- Saving data

Example:

```
public class AppDbContext : DbContext
```

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

    public DbSet<Product> Products { get; set; }
}
```

⭐ 4.2 DbSet

DbSet = **represents a database table**.

- DbSet<Product> → Products table
- Used to query and save instances of Product

Example Operations

```
context.Products.Add(new Product());    // Insert
context.Products.ToList();              // Select *
context.Products.Find(1);               // Get by Id
context.Products.Remove(product);        // Delete
context.SaveChanges();                  // Commit to DB
```

✅ 5. Code First Migrations

Migrations = EF Core’s way to **create**, **update**, and **sync** the database schema with your model classes.

Why do we need migrations?

Because models change over time:

```
public string Description { get; set; }
```

EF must update DB accordingly.

⭐ Migration Workflow

Step 1: Add Migration

```
Add-Migration InitialCreate
```

This creates a C# file containing SQL-like instructions.

Step 2: Apply Migration

```
Update-Database
```

EF creates tables in the DB.

⭐ Example Migration (auto-generated)

```
migrationBuilder.CreateTable(
  name: "Products",
  columns: table => new
  {
    Id = table.Column<int>(nullable: false)
      .Annotation("SqlServer:Identity", "1, 1"),
    Name = table.Column<string>(nullable: true),
    Price = table.Column<decimal>(nullable: false)
  },
  constraints: table =>
  {
    table.PrimaryKey("PK_Products", x => x.Id);
  });
```

⭐ Modify Model → Add New Migration

If you add:

```
public string Category { get; set; }
```

Run:

```
Add-Migration AddCategoryToProduct
Update-Database
```

EF adds a new column without losing data.

🎯 Final Summary

Concept	Meaning
EF Core	ORM tool for .NET to map classes ↔ tables
ORM	Converts C# objects into SQL & vice-versa
Data Providers	Allow EF to work with SQL Server, MySQL, PostgreSQL, etc.
Code First	Start with C# models → EF creates DB
DB First	Start with DB → EF generates classes
DbContext	Bridge between application and database
DbSet	Represents a table
Migrations	Track and update database schema