# Entity Framework Core

## Introduzione pratica

- Con Visual Studio 2022
- Demo completa: Modello, DbContext, Migrazioni, CRUD

# Che cos’è un ORM?

- **Definizione di base**
- Un **ORM (Object–Relational Mapper)** è una tecnologia che permette a un’applicazione di lavorare con un *database relazionale* usando **oggetti** invece che istruzioni SQL.
  L’ORM si occupa automaticamente della traduzione tra:
- **classi C# → tabelle del database**
- **proprietà → colonne**
- **istanze → righe**
- **LINQ → SQL generato automaticamente**
- In pratica, è un interprete instancabile che evita a te di scrivere SQL ripetito e noioso.

# Perché esiste un ORM

Gli ORM risolvono un problema storico:

il **mismatch tra mondo oggetti (OOP)** e **mondo relazionale (SQL)**

- In C# hai **classi, ereditarietà, metodi, oggetti**
- Nel database hai **tabelle, chiavi, righe, join**
- Due universi con logiche diverse, che non parlano la stessa lingua.
  L'ORM è il traduttore automatico.

# Cosa fa tecnicamente un ORM

1. **Mapping**
2. Tracking degli oggetti
3. Generazione automatica SQL
4. Gestione sicura delle Query
5. Migrazioni

### 1. Mapping

Associa classi/tabelle, proprietà/colonne, relazioni/fk.

Esempio:

```csharp
class Studente {
    public int Id { get; set; }
    public string Nome { get; set; }
}
```

diventa:

```markdown
Table Studenti
--------------------
Id (int, PK)
Nome (nvarchar)
```

# Cosa fa tecnicamente un ORM

1. Mapping
2. **Tracking degli oggetti**
3. Generazione automatica SQL
4. Gestione sicura delle Query
5. Migrazioni

**2. Tracking degli oggetti**

Tiene traccia degli oggetti caricati dal database:
capisce se sono **nuovi**, **modificati**, **da rimuovere**.

# Cosa fa tecnicamente un ORM

1. Mapping
2. Tracking degli oggetti
3. **Generazione automatica SQL**
4. Gestione sicura delle Query
5. Migrazioni

Quando fai:

```csharp
var elenco = db.Studenti.ToList();
```

l'ORM genera dietro le quinte qualcosa tipo:

```sql
SELECT [s].[Id], [s].[Nome], [s].[Eta]
FROM [Studenti] AS [s]
```

# Cosa fa tecnicamente un ORM

1. Mapping
2. Tracking degli oggetti
3. Generazione automatica SQL
4. **Gestione sicura delle Query**
5. Migrazioni

**4. Gestione sicura delle Query**

- Zero SQL injection.
- Zero concatenazioni di stringhe pericolose.
- Tutto tipizzato e controllato dal compilatore.

# Cosa fa tecnicamente un ORM

1. Mapping
2. Tracking degli oggetti
3. Generazione automatica SQL
4. Gestione sicura delle Query
5. **Migrazioni**

**5. Migrazioni**
Riesce a *creare* e *aggiornare* lo schema del database a partire dalle classi del modello.

# Vantaggi

- Meno codice SQL ripetitivo
- Query con LINQ, più espressive
- Modello dei dati coerente con il codice
- Portabilità tra database (EF Core supporta molti provider)
- Tempo di sviluppo ridotto

# Svantaggi

- Non è sempre ottimale per query ultra complesse
- Può generare SQL poco performante se mal usato
- Va compreso, non “uso e basta», senza analisi

# In sintesi

- Un ORM permette agli sviluppatori di lavorare con il database attraverso oggetti e classi
- Delega a un motore automatico la creazione delle query SQL e la gestione dei dati.
- Riduce complessità e codice, migliorando produttività e sicurezza.

# ARCHITETTURA EF CORE

**Spiegazione completa**

- L’architettura di EF Core è costruita a strati.
- La stratificazione evita che ad ogni cambiamento del database si debba riscrivere l'applicazione.

# 1. Application Layer
**(il nostro codice C#)**

## E‘ La nostra applicazione

- classi del modello (Studente, Corso, Docente…)
- servizi
- controller
- repository (se necessario)
- Program.cs

Qui tu **scrivi LINQ**, non SQL.

Esempio:

```
var elenco = db.Studenti.Where(s => s.Eta > 18).ToList();
```

# 2. DbContext
## (il cuore di EF Core)

È la **sessione** con il database, il componente centrale.

Si occupa di:

- tracking degli oggetti
- mapping tra classi e tabelle
- validazione
- gestione delle transazioni
- generazione delle query

È sempre DbContext che decide:

- quali entità sono *Added*
- quali sono *Modified*
- quali sono *Deleted*

Quando chiami il metodo SaveChanges(), parte tutto:
=> **EF Core** analizza la situazione e crea l'SQL necessario.

# 3. Change Tracker
## (tiene traccia di tutto )

Esempio:

```csharp
var stud = db.Studenti.First();
stud.Eta = 30;
```

Il Change Tracker pensa:

> "Ok, prima era 23, ora 30. Questo va aggiornato."

Quando fai `SaveChanges()`, genera:

```sql
UPDATE Studenti SET Eta = 30 WHERE Id = 1;
```

Così tu sembri potente, ma il lavoro duro lo fa lui.

# 4. Query Pipeline
## (LINQ → Expression Tree → SQL)

È il canale attraverso cui EF Core:

- Prende la tua query LINQ
- La trasforma in un **albero di espressioni**
- La ottimizza
- Chiede al provider di tradurla in SQL

Esempio:

```csharp
var studenti = db.Studenti.OrderBy(s => s.Nome).ToList();
```

Diventa:

```sql
SELECT [s].[Id], [s].[Nome], [s].[Eta]
FROM [Studenti] AS [s]
ORDER BY [s].[Nome]
```

# 5. EF Core Provider
## (SQL Server, SQLite, PostgreSQL, MySQL)

È il traduttore vero e proprio.
Senza provider, EF Core sarebbe rimarrebbe soltanto un concetto accademico.

I provider:

- traducono le query nel dialetto SQL corretto
- gestiscono i tipi di dato
- gestiscono funzionalità specifiche del DB
- eseguono le operazioni contro il database

Esempi:

- `Microsoft.EntityFrameworkCore.SqlServer`
- `Npgsql.EntityFrameworkCore.PostgreSQL`

# 6. Database relazionale

**(tutto si appoggia a un vecchio e caro RDBMS)**

- SQL Server
- MySQL
- PostgreSQL
- SQLite
- Oracle (quando esiste il provider)
- Il DB vede solo SQL.
  Non sa nulla delle tue classi, dei tuoi oggetti, dei tuoi LINQ.

# Flusso completo

- Noi scriviamo codice LINQ
- Il DbContext lo analizza
- Il Query Pipeline lo traduce in un albero di espressioni
- Il Provider genera SQL
- Il Database esegue
- EF Core rimappa i risultati in oggetti C#

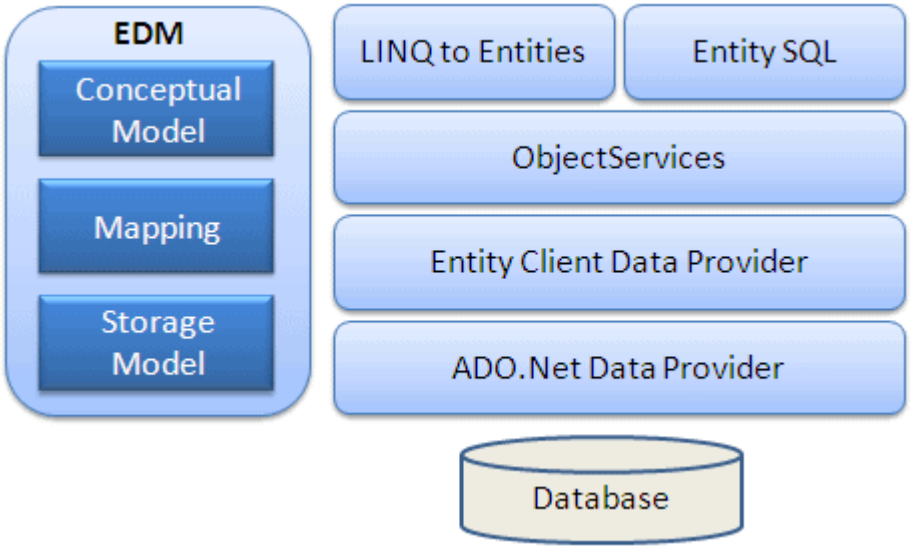
EF Core utilizza un’architettura a strati:
la nostra applicazione lavora con oggetti, il DbContext gestisce mapping e tracking, il provider traduce in SQL e il database esegue.

L’obiettivo è nascondere la complessità permettendoti di sviluppare più velocemente e con meno errori.

# Cos’è un ORM

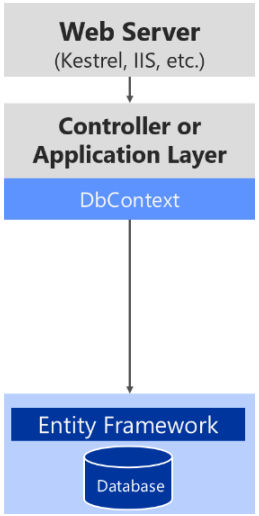
ORM = Object Relational Mapper

- Trasforma tabelle -> classi
- Meno SQL ripetitivo
- Query LINQ
- Modello coerente



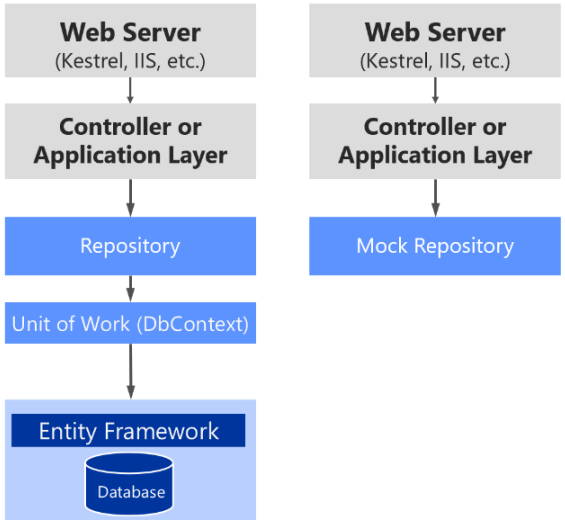
## No Repository

Direct access to database from controller




## With Repository

Abstraction layer between controller and database context.
Unit tests can mock data to facilitate testing



# Cos’è Entity Framework Core

- ORM moderno di .NET
- Open source
- Cross-platform
- Usa LINQ
- Supporto SQL Server, SQLite, PostgreSQL, MySQL

# Creazione progetto

Visual Studio 2022:

1. File → New → Project
2. Console App (.NET)
3. Nome: EfDemo
4. Framework .NET 6/7

# Installazione EF Core

Tools → NuGet Package Manager → Package Manager Console

Installare:

```
Install-Package Microsoft.EntityFrameworkCore
Install-Package Microsoft.EntityFrameworkCore.SqlServer
Install-Package Microsoft.EntityFrameworkCore.Tools
Install-Package Microsoft.EntityFrameworkCore.Design
```

# Creazione cartella Models

1. Add → New Folder → Models
2. Add → Class → Studente.cs

# Classe Studente

```
public class Studente {
    public int Id { get; set; }
    public string Nome { get; set; }
    public int Eta { get; set; }
}
```

Una classe POCO (Plain Old CLass) è una classe che rappresenta un semplice oggetto di dominio.  La sua semplicità la rende facilmente utilizzabile per serializzazione, deserializzazione e per passare dati tra diversi strati di un'applicazione.

# Creazione DbContext

```
public class ScuolaContext : DbContext {
    public DbSet<Studente> Studenti { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder options)
    {
       options.UseSqlServer("Server=localhost;Database=ScuolaDB;
        Trusted_Connection=True;TrustServerCertificate=True;");
    }
}
```

# Migrazioni

da Package Manager Console:

```
PM> Add-Migration Iniziale
PM> Update-Database
```

Risultato: Database ScuolaDB creato

# Verifica DB

View → SQL Server Object Explorer

Controllare esistenza tabella: Studenti

# Program.cs

## CRUD – Create & Read - (esempio)

```
using var db = new ScuolaContext();

// CREATE
db.Studenti.Add(new Studente { Nome = "Mario", Eta = 22 });
db.SaveChanges();

// READ
var elenco = db.Studenti.ToList();
foreach (var s in elenco)
    Console.WriteLine($"{s.Id} - {s.Nome} - {s.Eta}");

// UPDATE
var stud = db.Studenti.First();
stud.Eta = 23;
db.SaveChanges();

// DELETE
db.Studenti.Remove(stud);
db.SaveChanges();

Console.WriteLine("Operazioni completate.");
```

# CRUD – Update

```
var stud = db.Studenti.First();
stud.Eta = 23;
db.SaveChanges();
```

# CRUD – Delete

```
db.Studenti.Remove(stud);
db.SaveChanges();
```

# Struttura progetto

- EfDemo/
- ├── Program.cs
- ├── ScuolaContext.cs
- ├── Migrations/
- └── Models/Studente.cs

# Esercizio

- Creare entità Corso, Docente, Studente
- Relazioni 1:N e N:N
- CRUD completo

# Conclusioni

- ORM
- Modello
- DbContext
- Migrazioni
- CRUD con LINQ