

# Entity Framework

ORM di Microsoft basato sul .NET Framework

Insieme di tecnologie ADO.NET per lo sviluppo software

Definisce un modello di astrazione dei dati

Traduce il nostro codice in query comprensibili dal DBMS

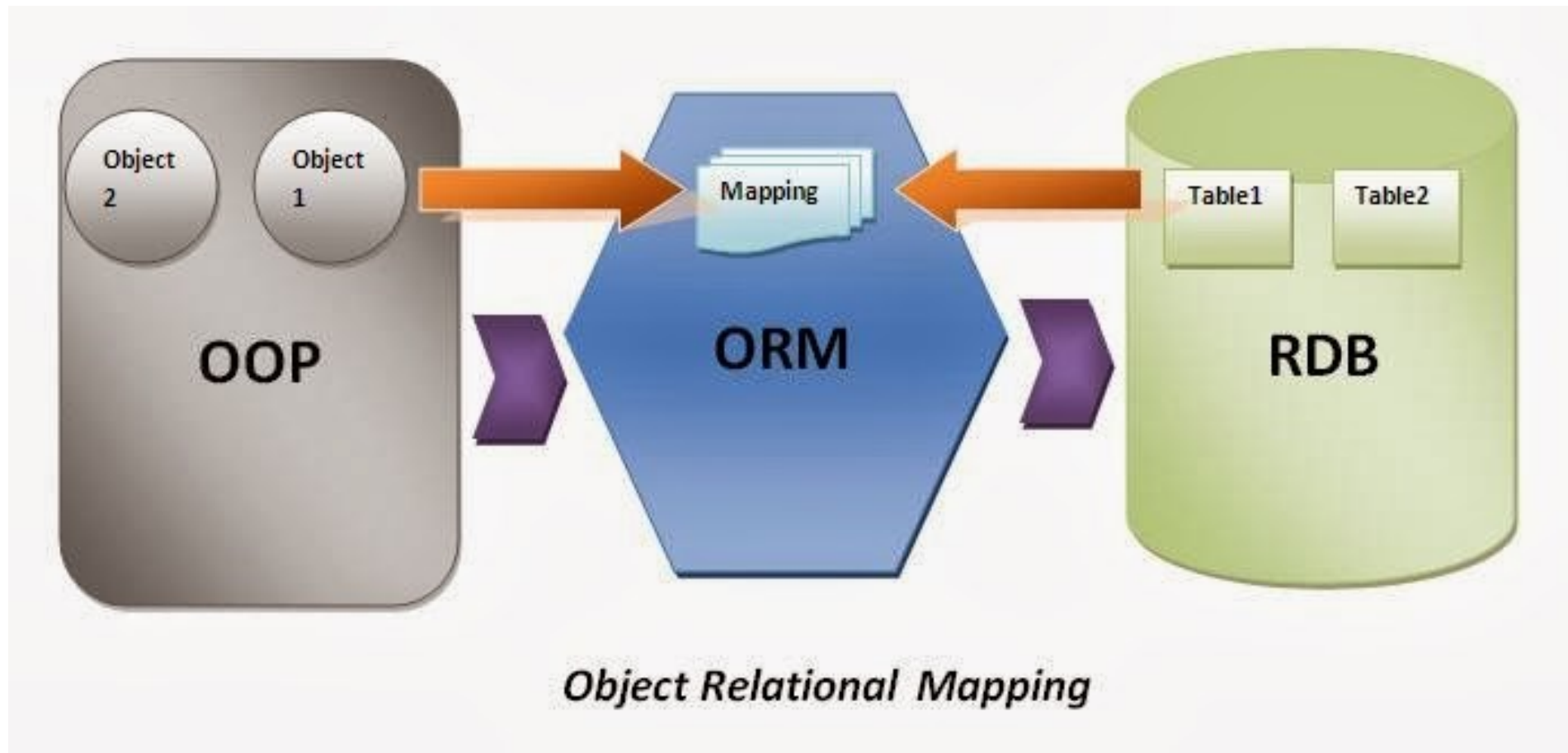
Disaccoppiamento tra applicazione e dati

- Posso mantenere la stessa rappresentazione anche se cambia il modello fisico (es. da SQL Server ad Oracle)

Open source

- <https://github.com/aspnet/EntityFramework>

# Cos'è un ORM?



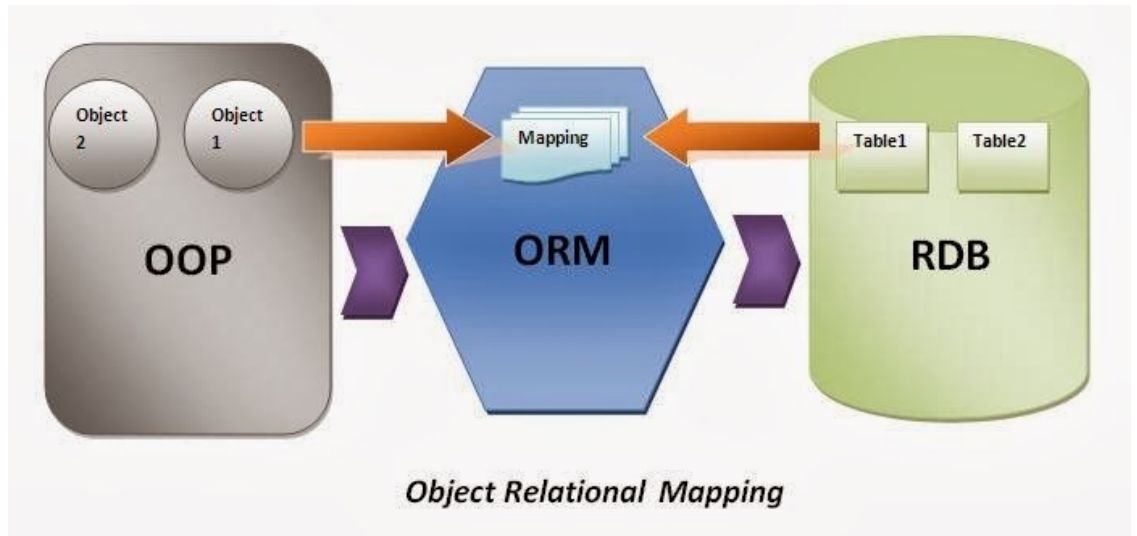
# Cos'è un ORM?

È una tecnica per convertire dati da type system incompatibili

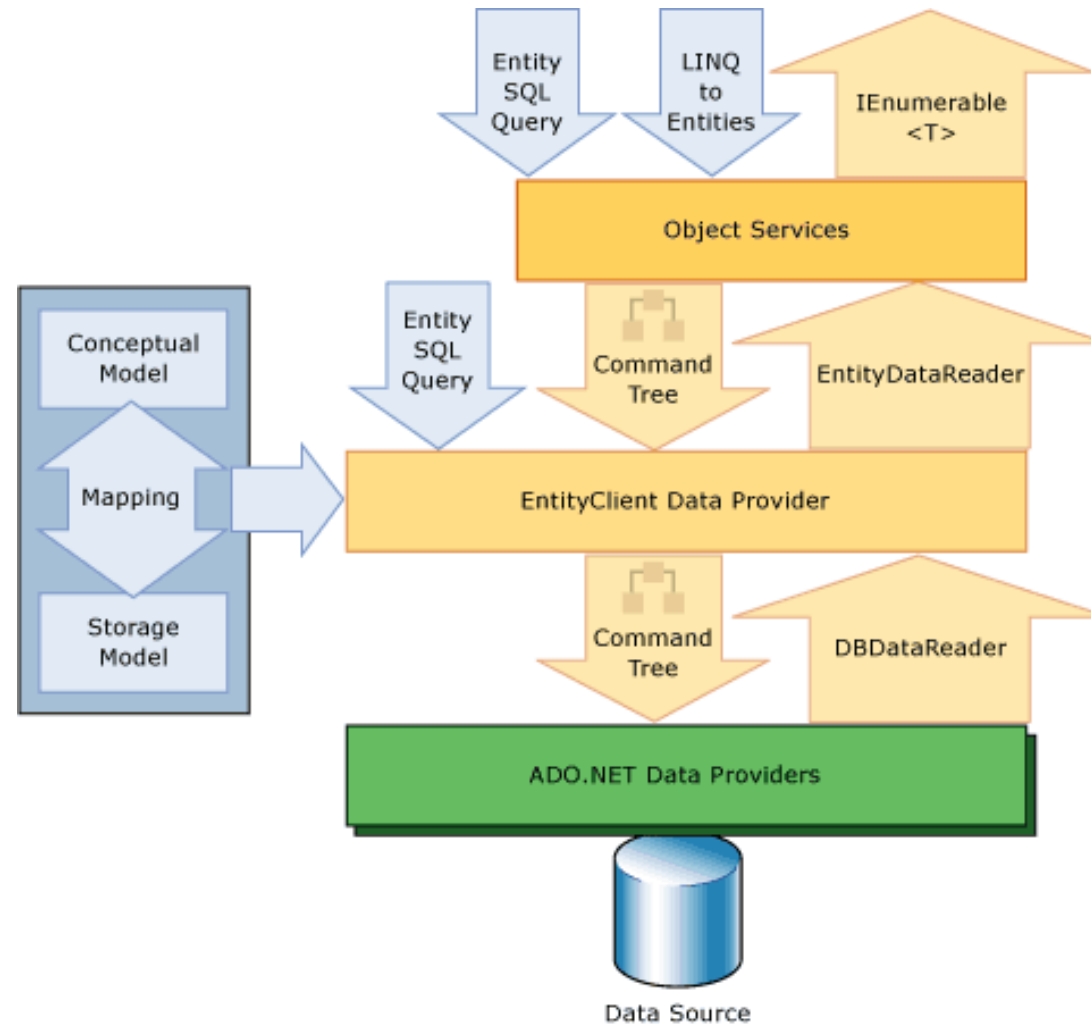
Da database ad object-oriented

3 caratteristiche fondamentali

- **Mapping**
  - Definisce come il database si «incastra» negli oggetti e viceversa
- **Fetching**
  - Sa come recuperare i dati dal database e materializzare i rispettivi oggetti
- **Persistenza del grafo**
  - Sa come salvare le modifiche agli oggetti, generando le query SQL corrispondenti



# Come funziona



# Diversi approcci

## Database-First

- Il modello viene importato da un DB esistente
- Se modifico il database posso (quasi) sempre aggiornare il modello

## Model-First

- Il modello del database viene creato dal designer di Visual Studio
- L'implementazione fisica è basata sul modello generato
- Non favorisce il riutilizzo del codice né la separazione tra contesto ed entità
- Poichè il modello definisce il DB, eventuali sue modifiche verranno perse

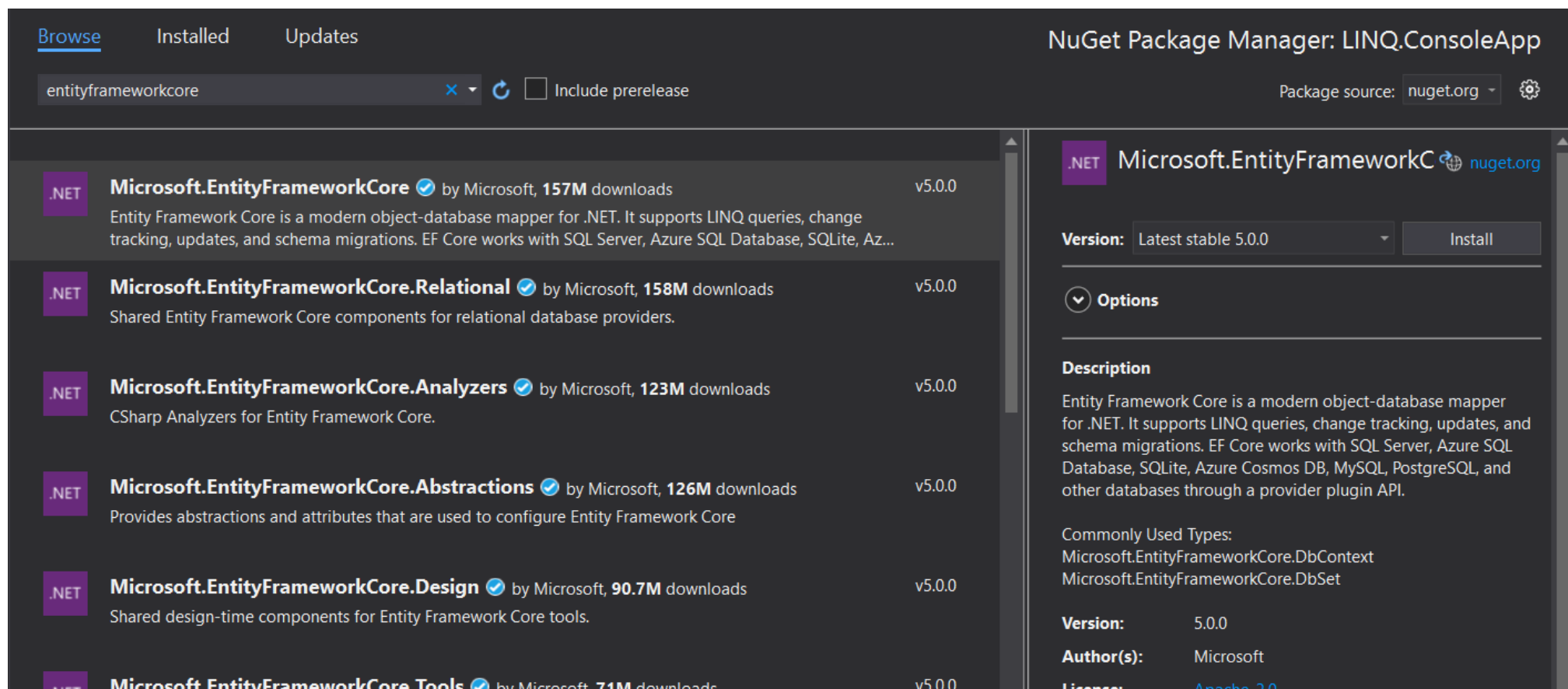
## Code-First

- Il modello viene creato dal nostro codice
- L'implementazione fisica è basata sul nostro codice

# Perché Code-First?

- Focus sul domain design
- C# potrebbe risultarci più familiare delle query di SQL
  - E sarebbe l'unico linguaggio da apprendere
- Possiamo mettere facilmente sotto source control il nostro database (niente script SQL solo codice C#)
- Evitiamo la mole di codice auto generato da EDMX
- Se scegliamo di sviluppare in .NET Core, l'EDMX non è supportato

# Configurazione di EF



The screenshot displays the NuGet Package Manager interface for a project named 'LINQ.ConsoleApp'. The search bar contains 'entityframeworkcore'. The results list several packages by Microsoft, all at version 5.0.0. The package 'Microsoft.EntityFrameworkCore' is selected, showing its description, options, and commonly used types.

**NuGet Package Manager: LINQ.ConsoleApp**

Package source: [nuget.org](https://nuget.org)

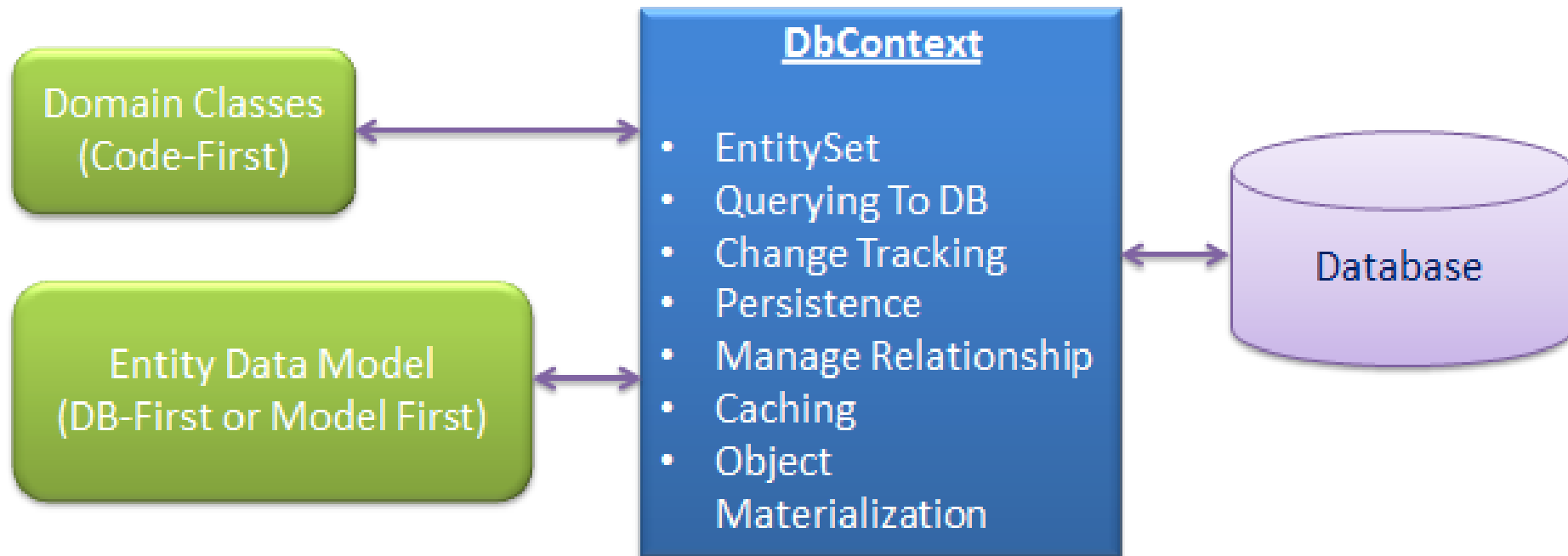
**Search Results:**

Package Name	Author	Downloads	Version
Microsoft.EntityFrameworkCore	Microsoft	157M	v5.0.0
Microsoft.EntityFrameworkCore.Relational	Microsoft	158M	v5.0.0
Microsoft.EntityFrameworkCore.Analyzers	Microsoft	123M	v5.0.0
Microsoft.EntityFrameworkCore.Abstractions	Microsoft	126M	v5.0.0
Microsoft.EntityFrameworkCore.Design	Microsoft	90.7M	v5.0.0
Microsoft.EntityFrameworkCore.Tools	Microsoft	71M	v5.0.0

**Microsoft.EntityFrameworkCore Details:**

- Version:** Latest stable 5.0.0
- Install** button
- Options** (expanded)
- Description:** Entity Framework Core is a modern object-database mapper for .NET. It supports LINQ queries, change tracking, updates, and schema migrations. EF Core works with SQL Server, Azure SQL Database, SQLite, Azure Cosmos DB, MySQL, PostgreSQL, and other databases through a provider plugin API.
- Commonly Used Types:**
  - Microsoft.EntityFrameworkCore.DbContext
  - Microsoft.EntityFrameworkCore.DbSet
- Version:** 5.0.0
- Author(s):** Microsoft
- License:** Apache 2.0

# II DbContext





## II DbContext

```
public class Context : DbContext
{
    public DbSet<Student> Students { get; set; }

    public Context() : base() { }

    public Context() : base("MyContext") { }
}
```

# II DbContext

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

```
{
    // ...
    "ConnectionStrings": {
        "TicketingDb": "Server=tcp:democrito.database.windows.net,1433;Initial Catalog=Ticketing;
            Persist Security Info=False;User ID=sa;Password=xxxxxxx;MultipleActiveResultSets=False;
            Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;"
    }
}
```

# Il DbSet

E' una classe che rappresenta le entity

Serve per fare operazioni CRUD

E' definito come **DbSet<TEntity>**

I metodi più utilizzati sono:

- **Add, Remove, Find, SqlQuery**

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

# Type Discovery

Nel *DbContext*:

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

Definizione della classe *Student*:

```
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime DateOfBirth { get; set; }

    public Teacher Teacher { get; set; }
}
```

# Primary Key

```
public class Student
{
    public int StudentID { get; set; }

    public string StudentName { get; set; }
    public DateTime DateOfBirth { get; set; }
    public Teacher Teacher { get; set; }
}
```

Convenzione sul nome della chiave primaria:

- *Id*
- *<NomeClasse>ID*

```
public class Student
{
    public int MyPrimaryKey { get; set; }

    public string StudentName { get; set; }
    public DateTime DateOfBirth { get; set; }
    public Teacher Teacher { get; set; }
}
```

Non usa la convenzione di code-first  
Genera una *ModelValidation Exception*  
se non gestita con le *DataAnnotations*

# Foreign Key

```
public class Student
{
    public int StudentID { get; set; }

    public string StudentName { get; set; }
    public DateTime DateOfBirth { get; set; }

    public int CourseId { get; set; }
    public Course Course { get; set; }
}
```

```
public class Course
{
    public int CourseId { get; set; }

    public string CourseName { get; set; }
    public Teacher Teacher { get; set; }
    public ICollection<Student> Students { get; set; }
}
```

La *ForeignKey* viene generata automaticamente da code-first ogni volta che viene individuata una navigation property

Sempre bene rispettare le stesse convenzioni della *PrimaryKey*

# Navigation Properties

Cos'è la proprietà Teacher?

```
public class Student
{
    // ...
    public Teacher Teacher { get; set; }
}
```

È una **Navigation Property**.

È la rappresentazione in EF di una Relazione tra due entità.

# DataAnnotations e Fluent API

Le DataAnnotations sono attributi che servono a specificare il comportamento per fare l'override delle convenzioni di code-first

Possono influenzare le singole proprietà

- Namespace *System.ComponentModel.DataAnnotations*
- *Key, Required, MaxLength...*

Possono influenzare lo schema del database

- Namespace *System.ComponentModel.DataAnnotations.Schema*
- *Table, Column, NotMapped...*

Le DataAnnotations sono limitate. Per il set completo bisogna andare di Fluent API



# DataAnnotations: Key

Override della convenzione sulla *PrimaryKey*

Viene applicato alle proprietà di una classe

```
[Key]  
public int MyPrimaryKey { get; set; }
```

# DataAnnotations: Required

Indica al database che quella colonna non può essere NULL  
In ASP.NET MVC viene usato anche per la validazione

```
[Required]  
public string Name { get; set; }
```

# DataAnnotations: MaxLenght, MinLenght

Possono essere applicati a stringhe o array

Possono essere usati in coppia

*EntityValidationError* se non rispettati durante una update

```
[MaxLenght(50), MinLenght(8)]  
public string Name { get; set; }
```

# DataAnnotations: Table

Rappresenta l'override del nome della tabella

Può essere solo applicato ad una classe e non alle proprietà

Si può anche inserire uno schema differente

```
[Table("Studente", Schema = "MySchema")]  
public class Student { /* ... */ }
```

# DataAnnotations: Column

Rappresenta l'override del nome della proprietà

Può essere applicato solo ad una proprietà

Si può usare in combinata con *Order* e *TypeName*

```
[Column("Nome", Order = 5, TypeName = "varchar")]  
public string Name { get; set; }
```

# DataAnnotations: ForeignKey

Rappresenta l'override della convenzione sulla chiave esterna

Viene applicato solo alle proprietà di una classe

```
public class Student
{
    public int StudentId { get; set; }
    public int CourseId { get; set; }

    [ForeignKey("CourseId")]
    public Course Course { get; set; }
}

public class Course
{
    public int CourseId { get; set; }
    public ICollection<Student> Students { get; set; }
}
```

# DataAnnotations: NotMapped

Ignora il mapping per proprietà che hanno getter e setter impostati  
Viene usato per non creare colonne nel database

```
[NotMapped]  
public string Name { get; set; }
```

# Fluent API

Sono una alternativa completa alle DataAnnotations  
Si definiscono dentro l'override di *OnModelCreating*

Tre tipologie di mapping supportate:

- **Model**: Schema e convenzioni
- **Entity**: Ereditarietà
- **Property**: chiavi primarie/esterne, colonne e altri attributi



# Model e Entity Mapping

Configurazione dello schema per tutto il database

Configurazione dello schema per singola tabella

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.HasDefaultSchema("Admin");

    //Map entity to table
    modelBuilder.Entity<Student>().ToTable("StudentInfo");
    modelBuilder.Entity<Student>().ToTable("StandardInfo", "anotherSchema");
}
```

# Property Mapping

Configurazione della chiave primaria

```
modelBuilder.Entity<Student>().HasKey<int>(s => s.StudentId);
```

Configurazione di altre proprietà

```
modelBuilder.Entity<Student>().Property(p => p.Age)
    .HasColumnName("Eta")
    .HasColumnOrder(3)
    .HasColumnType("datetime")
    .IsRequired();
```

# Fluent API Configurations

Tutte le configurazioni sono fatte via Fluent API

- Problema: troppo codice dentro *OnModelCreating*, diventa ingestibile!
- Soluzione: organizziamo le configurazioni

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.ApplyConfigurations<Student>(new StudentConfiguration());
    modelBuilder.ApplyConfigurations<Course>(new CourseConfiguration());
}
```

# Fluent API Configurations

```
public class StudentConfiguration : IEntityTypeConfiguration<Student>
{
    public void Configure(EntityTypeBuilder<Student> builder)
    {
        builder.ToTable("StudentInfo");

        builder.HasKey<int>(s => s.StudentId);

        builder.Property(p => p.Age)
            .HasColumnName("Eta")
            .HasColumnOrder(3)
            .HasColumnType("datetime")
            .IsRequired();
    }
}
```

# Relazioni uno-a-uno

Una relazione uno-a-uno è, per definizione, una relazione per cui la chiave primaria di una tabella diventa chiave primaria e chiave esterna dell'altra

```
public class Student
{
    [Key]
    public int StudentId { get; set; }
    public string StudentName { get; set; }

    public virtual Address Address { get; set; }
}
```

```
public class Address
{
    public string Address { get; set; }
    public string City { get; set; }

    [Key, ForeignKey("Student")]
    public int StudentId { get; set; }
    public virtual Student Student { get; set; }
}
```

# Relazioni uno-a-uno

Si può fare anche da Fluent API

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {  
    modelBuilder.Entity<StudentAddress>().HasKey(e => e.StudentId);  
  
    modelBuilder.Entity<Student>()  
        .HasOptional(s => s.StudentAddress)  
        .WithRequired(ad => ad.Student);  
}
```

# Relazioni uno-a-molti

Scenario: un insegnante può tenere più di un corso

```
public class Teacher
{
    public Teacher()
    {
        Courses = new List<Course>();
    }

    [Key]
    public int TeacherId { get; set; }
    public string FullName { get; set; }
    public virtual ICollection<Course> Courses { get; set; }
}
```

```
public class Course
{
    [Key]
    public int CourseId { get; set; }
    public string CourseName { get; set; }
    public int TeacherId { get; set; }

    public virtual Teacher Teacher { get; set; }
}
```

# Relazione uno-a-molti

Si può fare anche da Fluent API

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Course>()
        .HasRequired<Teacher>(s => s.Teacher)
        .WithMany(s => s.Courses)
        .HasForeignKey(s => s.TeacherId);

    modelBuilder.Entity<Teacher>()
        .HasMany(s => s.Courses)
        .WithRequired(x => x.Teacher)
        .HasForeignKey(x => x.TeacherId);
}
```



# Relazioni multi-a-molti

Scenario: uno studente è iscritto a più corsi e ogni corso può avere più studenti

```
public class Course
{
    [Key]
    public int CourseId { get; set; }
    public string CourseName { get; set; }
    public virtual ICollection<Student> Students { get; set; }
}
```

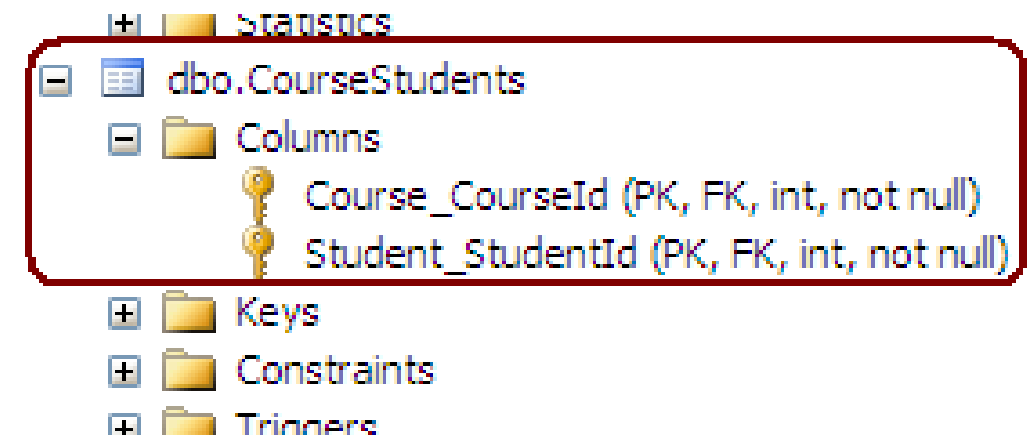
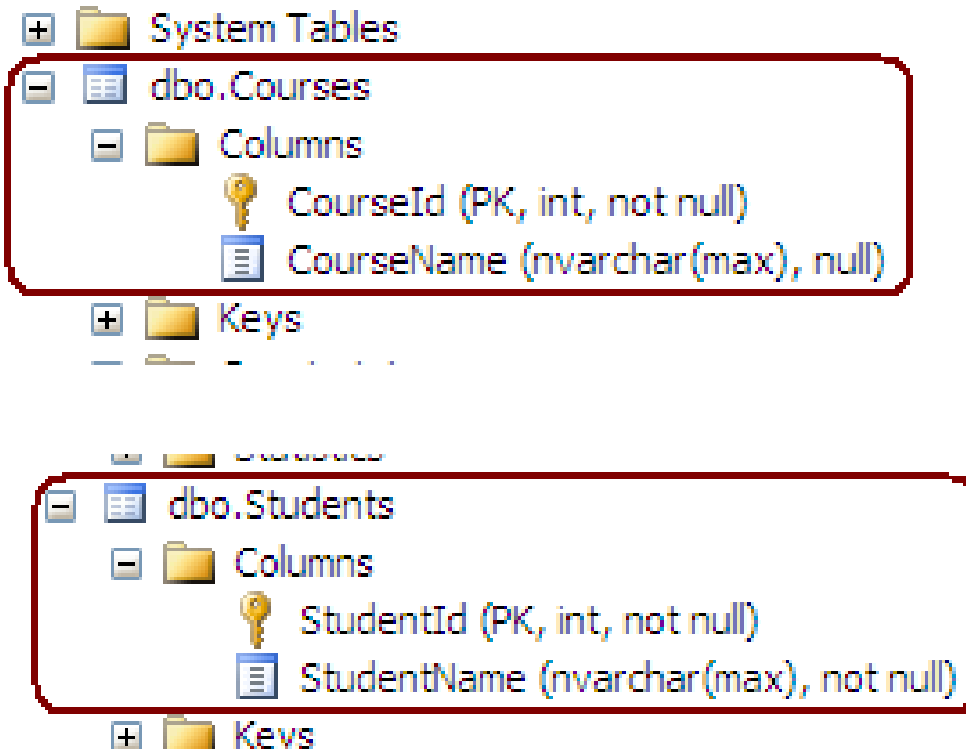
# Relazioni multi-a-molti

Scenario: uno studente è iscritto a più corsi e ogni corso può avere più studenti

```
public class Student
{
    public Student()
    {
        Courses = new List<Course>();
    }

    [Key]
    public int StudentId { get; set; }
    public string StudentName { get; set; }
    public virtual ICollection<Course> Courses { get; set; }
}
```

# Relazioni multi-a-molti



# Aggiungere dati

Utile sapere:

- Il pattern *IDisposable*
- *Async/await*

```
using (var ctx = new Context())  
{  
    var person = new Person("Matteo", "Tumiati");  
    ctx.People.Add(person);  
    await ctx.SaveChangesAsync();  
}
```

# Fare query sui dati

Utile conoscere:

- Il pattern *IDisposable*
- *Linq*
- I dati che si vogliono ottenere ☺

```
using (var ctx = new Context())  
{  
    ctx.Students.Where(x => x.Name.StartsWith("A"))  
                .OrderBy(x => x.Age)  
                .ToList();  
}
```

# Errori comuni

Giusto

```
var ages = dbContext.People
    .Where(x => x.LastName.StartsWith("A"))
    .OrderBy(x => x.Age)
    .Where(x => x.City == "Bologna")
    .ToList();
```

Sbagliato

```
var ages = dbContext.People
    .Where(x => x.LastName.StartsWith("A"))
    .ToList()
    .OrderBy(x => x.Age)
    .Where(x => x.City == "Bologna")
    .ToList();
```

# Errori comuni

Giusto

```
var person = dbContext.People.Find(1);
```

Quasi giusto 😊

```
var person = dbContext.People  
    .Where(x => x.Id == 1);
```

# CUD (Create / Update / Delete)

Connected

Inserimento

```
using(var ctx = new MyContext())
{
    var prd = new Product()
    {
        ProductCode = "PR0001"
    };

    ctx.Products.Add(prd);

    ctx.SaveChanges();
}
```



# CUD (Create / Update / Delete)

## Connected

### Update / Delete

```
using(var ctx = new MyContext())
{
    var prd = ctx.Products.FirstOrDefault<Product>();

    // UPDATE
    prd?.ProductCode = "PR0002";
    ctx.SaveChanges();

    // DELETE
    if(prd != null)
        ctx.Products.Remove(prd);
    ctx.SaveChanges();
}
```

# Query Home-Made

Si può bypassare uno strato di Entity Framework che si occupa della traduzione della query da Linq to Entities

```
var people = dbContext.Database.SqlQuery<Person>(
    "SELECT * FROM Person WHERE FirstName LIKE 'a%')
.ToList();
```

# Caricamento dei Dati Correlati

Entity Framework Core consente di utilizzare le Navigation Property nel modello per caricare entità correlate

Esistono due modelli comuni utilizzati per caricare i dati correlati:

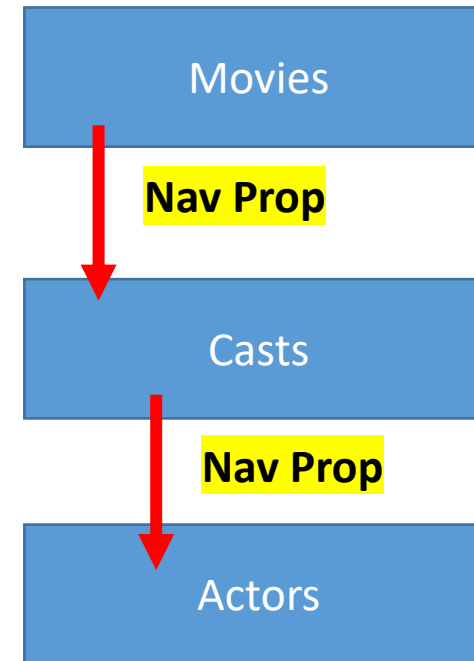
- **Eager Loading:** i dati correlati vengono caricati dal database come parte della query iniziale
- **Lazy Loading:** i dati correlati vengono caricati in modo trasparente dal database quando si accede alla proprietà di navigazione

# Caricamento dei Dati Correlati

## Eager Loading

È possibile utilizzare il metodo **Include** per specificare i dati correlati da includere nei risultati della query

```
using var ctx = new MyContext();  
  
var data = ctx.Movies  
    .Include(m => m.Casts)  
    .Include(c => c.actors)  
    .ToList();
```



# Migrations

Sono il meccanismo che consente, utilizzando l'approccio code-first, l'aggiornamento del database a fronte di modifiche al modello.

Esistono 2 tipi di migrazioni

- *Automatiche*: poco invasive
- *Manuali o code-based*: richiedono un intervento specifico sul database

# Migrazioni code-based

Sono utili quando vogliamo più controllo sulle modifiche automatiche. Servono due comandi dalla Package Manager Console:

- **Add-Migrations «Migration name»**
  - Crea una nuova classe con tutte le modifiche rispetto allo stato precedente del db
- **Update-Database**
  - Aggiorna il database sulla base del modello

Si può anche fare rollback di una modifica:

- **Update-Database -Migration:"Migration name"**

# Migrazioni code-based

Per ogni Migration, viene creato un nuovo file per ogni migrazione

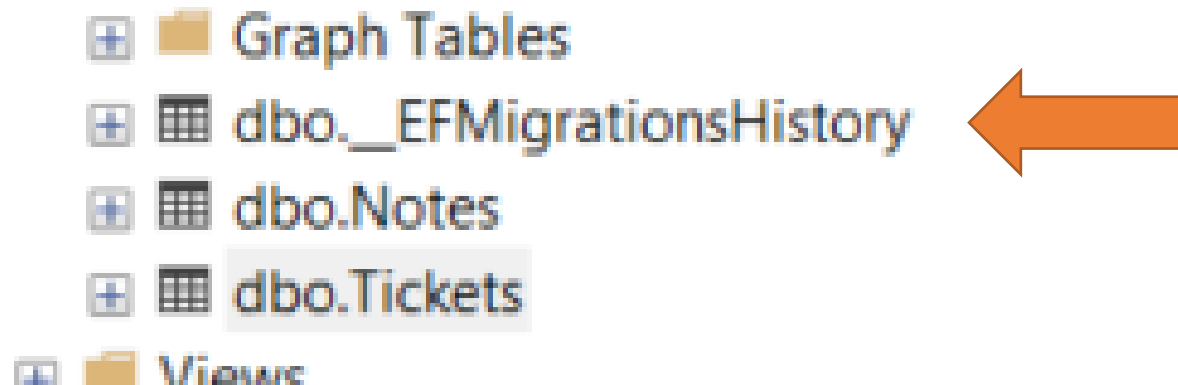
- `TimeStamp + NomeMigrazione.cs`
- Il file contiene una classe che eredita da **Migration**
- Contiene due metodi **Up** e **Down** per l'aggiornamento del database

```
public partial class FirstMigration : DbMigration
{
    public override void Up()
    {
        CreateTable("dbo.Students", c => new {
            StudentId = c.Int(nullable: false, identity: true),
            Name = c.String(),
            Age = c.Int(nullable: false)})
            .PrimaryKey(t => t.StudentId);
    }

    public override void Down()
    {
        DropTable("dbo.Students");
    }
}
```

# Migrazioni

Se andiamo a vedere il nostro database...



Viene aggiunta una tabella al nostro database per mantenere lo storico delle Migration applicate.