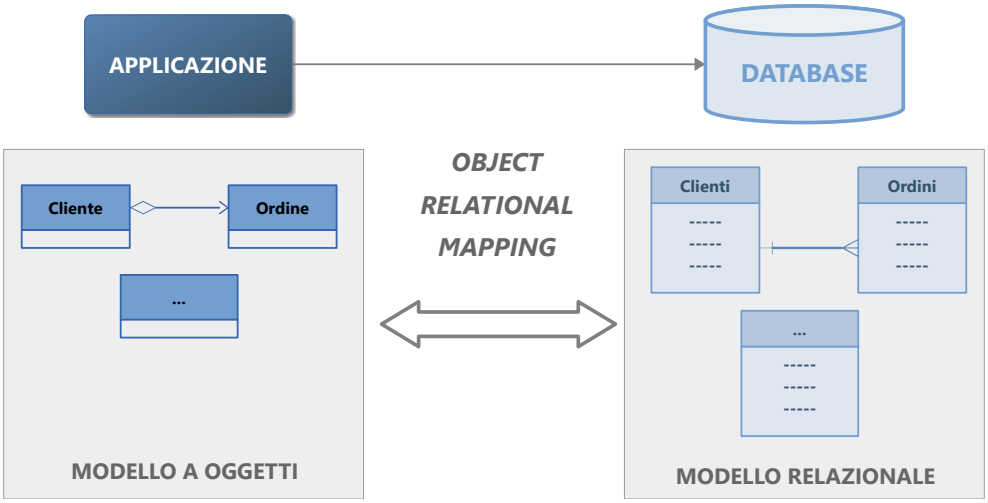
# ENTITY FRAMEWORK (Core)

**Anno 2022/2023**

## Indice generale

# 1 Object-Relational Mapping

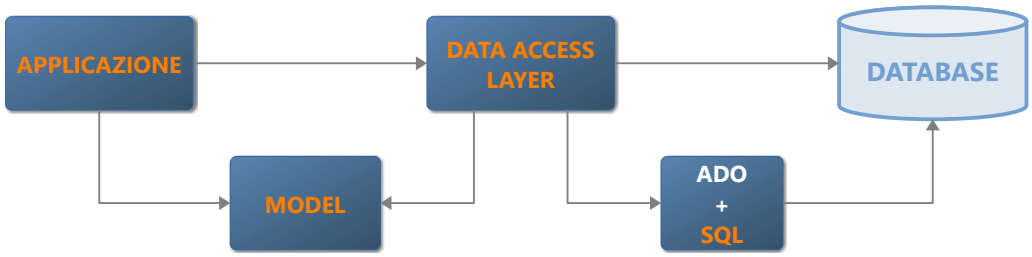
Esiste una problematica generale che riguarda la maggior parte di applicazioni che usano i database: far corrispondere il *modello a oggetti* usato nell'applicazione con il *modello relazionale* del database. Si parla di *Object-Relational Mapping*:




Nel *modello a oggetti* le *entità* e le loro *associazioni* sono rappresentate mediante classi. Nel *modello relazionale* le entità sono implementate mediante tabelle e le associazioni mediante le corrispondenze chiavi primarie ↔ chiavi esterne.

Una soluzione standard a questa problematica prevede la realizzazione di uno "strato" di software (*layer*) che fa da ponte tra il *modello a oggetti* e il database e fornisce i servizi richiesti, dall'esecuzione delle query alla modifica dei dati, implementando l'*Object-Relational Mapping*.

Questo *layer* utilizza il linguaggio SQL e una specifica tecnologia di accesso al database, che in .NET è ADO.NET (*ActiveX Data Object*):




Il problema di questa soluzione è che richiede la scrittura e la manutenzione di molte classi, con tutte le problematiche che questo comporta.

Esiste però un'alternativa che delega l'implementazione del *data access layer* a un software dedicato: un ***Object-Relational Mapper***.

## 1.1 Object-Relatonal-Mapper

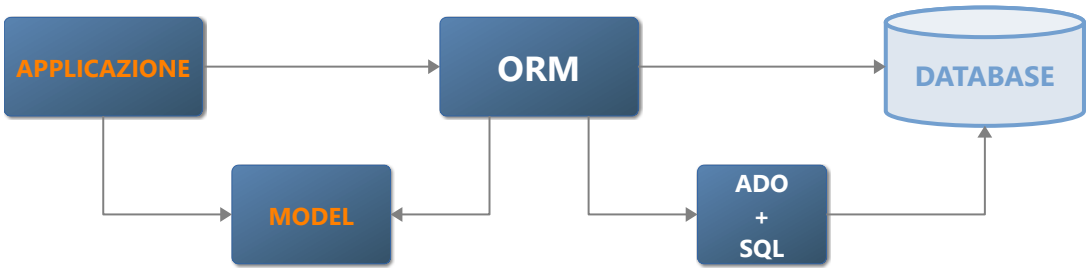
Con il termine ORM (***Object-Relational Mapper***) ci si riferisce a:

> *un'API orientata agli oggetti che favorisce l'integrazione tra applicazioni e sistemi DBMS e fornisce i servizi inerenti la persistenza dei dati, astraendo le caratteristiche implementative dell'RDBMS utilizzato.*[1]

Un ORM fornisce un livello di astrazione interposto tra l'applicazione e il database che:

1. Consente di creare una corrispondenza il modello *object oriented* dell'applicazione e il modello relazionale del database.
2. Semplifica le operazioni di interrogazione e manipolazione dei dati.
3. Implementa l'indipendenza dall'origine dei dati: cambiare DBMS non implica modificare il codice che lo usa.
4. Semplifica lo sviluppo di architetture *N-tier* (a più livelli).

In sintesi, un ORM evita al programmatore di dover scrivere il software di accesso ai dati, istruzioni SQL comprese:




## 1.2 Funzione fondamentale di un ORM

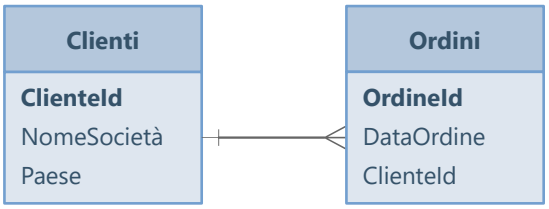
Alla base del funzionamento di un ORM c'è la capacità di "mappare" gli oggetti del modello con le tabelle e le relative associazioni. Ciò consente al codice applicativo di utilizzare il *modello a oggetti* senza doversi preoccupare di dove e in che modo i dati siano persistiti.

> **ORM e *storage* dei dati**
>
> Qui si da per scontato che l'ORM debba dialogare con un DBMS, ma non è detto che sia così. Uno dei vantaggi nell'utilizzo di un ORM è quello di potersi concentrare sul *modello a oggetti* indipendentemente dall'origine dei dati.

1 Da Wikipedia: "http://it.wikipedia.org/wiki/Object-relational_mapping"

### 1.2.1 Esempio di mapping tra object model e database

Per capire il ruolo svolto da un ORM, considera un database per la gestione degli ordini; più precisamente le tabelle **Clienti** e **Ordini**:




Considera la necessità di ottenere l'elenco dei clienti italiani e degli ordini che hanno eseguito. In risposta alla query il DBMS fornisce un *result set*, cioè un insieme tabellare di righe:

| Clienti.Client | NomeSocieti | Paese | OrdineId | DataOrdine | Ordini.Client |
|---|---|---|---|---|---|
| 1 | SuperSport | Italia | 4 | 10/11/2013 | 1 |
| 1 | SuperSport | Italia | 2 | 06/11/2013 | 1 |
| 3 | Sport time | Italia | 5 | 12/11/2013 | 3 |

Nell'applicazione, però, si vuole programmare in termini di classi:

```
public class Cliente
{
    public int ClienteId { get; set; }
    public string NomeSocietà { get; set; }
    public List<Ordine> Ordini { get; set; }
}

public class Ordine
{
    public int OrdineId { get; set; }
    public DateTime DataOrdine { get; set; }
    public Cliente Cliente { get; set; }
}
```

Questo ci consente scrivere il seguente codice:

```
var clienti = ... // ottieni i clienti dal database
foreach (var cliente in clienti.Where(c => c.Paese == "Italia"))
{
    foreach (var ordine in cliente.Ordini)
    {
        Console.WriteLine(ordine.DataOrdine); // -> date degli ordini dei clienti italiani
    }
}
```

Per rendere tutto ciò possibile è necessario interrogare il database e creare gli oggetti, "popolandoli" con i dati ottenuti dal *result set*. Un ORM è in grado di eseguire automaticamente queste ed altre operazioni.

## 1.3 Caratteristiche generali degli ORM

Gli ORM offrono diverse funzioni:

1. Generazione automatica del *modello a oggetti* sulla base dello schema del database.
2. Generazione del database a partire dal *modello a oggetti*. (Aggiornamento dello schema del database in relazione ai cambiamenti del *modello a oggetti*.)
3. Un linguaggio di interrogazione in grado di operare sul *modello a oggetti* e di agire in modo trasparente sul database.
4. Gestione della concorrenza.
5. Pattern ***Unit of Work***: eseguire un insieme di modifiche come un'unità, che deve essere completata con successo, oppure essere completamente annullata in caso di errore.

## 1.4 ORM più utilizzati

Restando in ambiente .NET, esistono diversi ORM che offrono funzionalità comparabili, tra i quali:

| | Entity Framework | Hibernate | DevExpress | Devart |
|---|---|---|---|---|
| Open source | SI | SI | NO | NO |
| Multi SO | SI | SI | SI | SI |
| **Linguaggi supportati** | | | | |
| C# | SI | Si | SI | Si |
| Java | NO | Si | NO | Si |
| **Database supportati**[2] | | | | |
| SQL Server | SI | SI | SI | SI |
| Oracle | SI | SI | SI | SI |
| MySql | SI | SI | SI | SI |
| SQLite | SI | SI | SI | SI |
| Access | NO | NO | SI | SI |

2 Sono riportati soltanto alcuni dei database supportati.

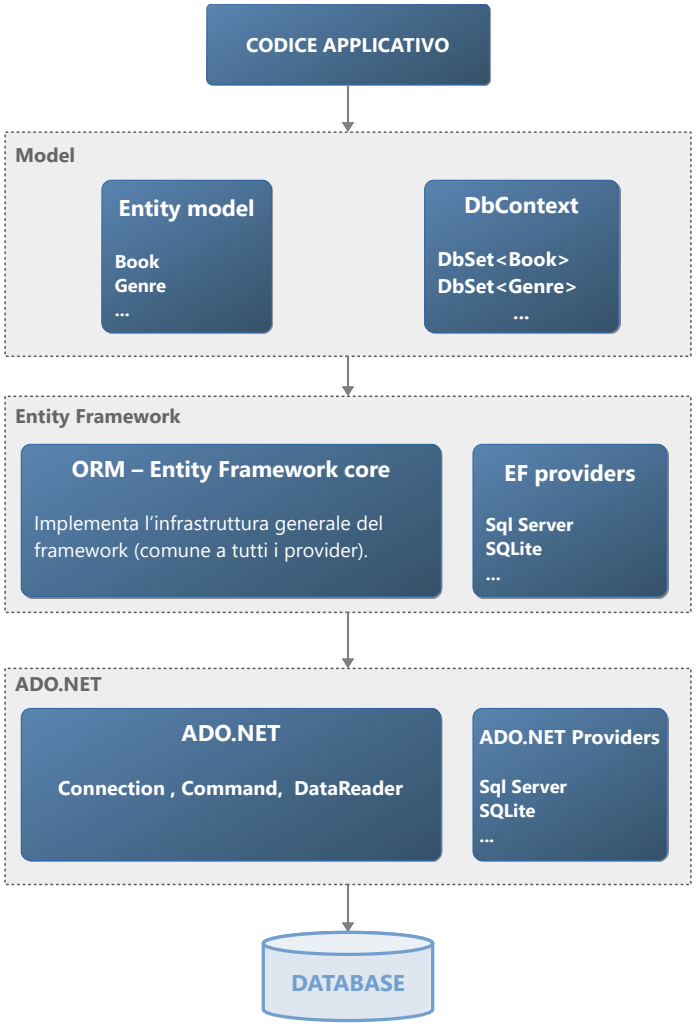
# 2 Introduzione a Entity Framework

Entity Framework (d'ora in avanti EF) è sviluppato da Microsoft e scaricabile come pacchetto NuGet.

> **EF 6 e EF Core**
>
> Esistono due implementazioni "parallele" di Entity Framework: **EF 6**, legato alla vecchia piattaforma .NET Framework e utilizzabile soltanto in ambiente Windows; **EF Core**, legato alla piattaforma .NET Core e utilizzabile in vari sistemi operativi.
>
> In questo tutorial faccio riferimento alla versione **EF Core**, ma le funzioni mostrate sono utilizzabili anche con **EF 6**.

## 2.1 Panoramica generale su EF

EF astrae le operazioni con il database consentendo al codice applicativo di agire sul *modello a oggetti* (chiamato anche *entity model*) e producendo i comandi SQL necessari. Le operazioni verso il database si avvalgono delle funzioni fornite da ADO.NET.



### 2.1.1 EF provider

Un ***EF provider*** è uno "strato" di software che si interpone tra EF e il DBMS con il quale deve dialogare. Dunque, EF non è utilizzabile con quei DBMS per i quale non esiste un *EF provider*. (Ad esempio, non esiste un *EF provider* per Access.)

Gli *EF provider* si differenziano per il livello di supporto che forniscono, cosa che dipende soprattutto dalle caratteristiche del DBMS utilizzato. Ad esempio, se il DBMS non gestisce le transazioni, nemmeno il *provider* sarà in grado di farlo.

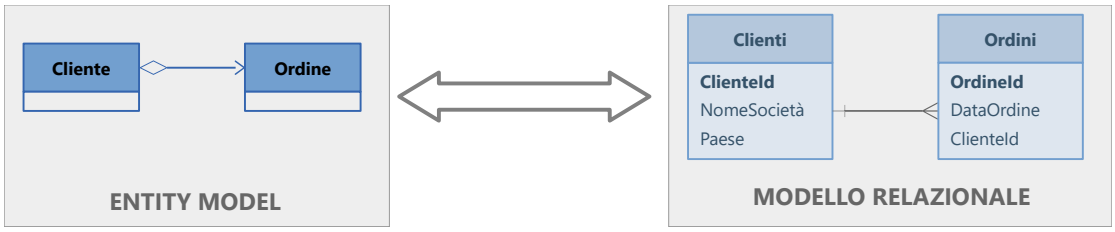
Altre differenze tra gli *EF provider* riguardano il supporto che forniscono a *design-time*:

1. Capacità di progettare il *domain model*.
2. Possibilità di creare il modello a partire dal database.
3. Possibilità di sincronizzare il database e il *domain model*, etc.

## 2.2 Entity model ed entity class

L'***entity model*** è alla base di tutte le applicazioni che usano EF e ha una corrispondenza con il modello *entity-relationship*. Ogni classe dell'*entity model* viene convenzionalmente definita ***entity class*** e, normalmente, corrisponde a una tabella del database.

Considera un semplice database composto dalle tabelle **Ordini** e **Clienti**. L'*entity model* corrispondente deve essere composto da due classi che corrispondono alle tabelle:




```
public class Cliente
{
    public int ClienteId { get; set; }
    public string NomeSocietà { get; set; }
    public List<Ordine> Ordini { get; set; }
}

public class  Ordine
{
    public int OrdineId { get; set; }
    public DateTime DataOrdine { get; set; }
    public Cliente Cliente { get; set; }
}
```

### 2.2.1 Requisiti delle entity class

Le *entity class* devono soddisfare dei requisiti:

- Devono essere pubbliche.
- Devono usare delle proprietà (non dei campi) per i valori corrispondenti alle colonne della tabella.
- Deve esistere una ***proprietà di chiave primaria***; cioè una proprietà omologa alla chiave primaria della tabella corrispondente.
- Devono avere un *costruttore predefinito*.

### 2.2.2 Entità

Nella terminologia comune, un oggetto dell'*entity model* viene definito **entità**. Un'*entità* è dunque un'istanza di un'*entity class* e rappresenta l'analogo del record di una tabella.

## 2.3 DbContext

La classe `DbContext` rappresenta il punto d'accesso a tutte le funzioni fornite da EF. Un oggetto `DbContext` (semplicemente *context*) gestisce tutte le operazioni sulle *entità*, traducendole in istruzioni SQL dirette verso il database.

### 2.3.1 Definire una classe "context"

La classe `DbContext` non viene usata direttamente; si definisce una classe derivata che faccia riferimento a uno specifico *entity model* e, attraverso una *stringa di connessione*, a uno specifico database.

Supponi di avere un database di nome **Library** che definisce le tabelle **Genres** e **Books**., tra le quali esiste un'associazione 1-N. Per gestirlo con EF occorre innanzitutto definire un *entity model* adeguato. Ad esempio:

```
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public int GenreId { get; set; }
    public Genre Genre { get; set; }
}

public class Genre
{
    public int GenreId { get; set; }
    public string Name { get; set; }
}
```

Quindi è necessario implementare una classe *context* che crei una corrispondenza tra le classi dell'*entity model* e le tabelle del database. Tale corrispondenza è definita attraverso delle proprietà di tipo `DbSet<>`.

### 2.3.2 DbSet<>: “mappare” le tabelle

Le proprietà `DbSet<>` rappresentano gli elementi centrali della classe *context*, poiché forniscono l’accesso alle tabelle del database: ad ogni proprietà *dbset* corrisponde una tabella. Ad esempio:

```
using Microsoft.EntityFrameworkCore;

class Library: DbContext
{
    public DbSet<Book> Books { get; set; }         // corrisponde alla tabella Books
    public DbSet<Genre> Genres { get; set; }       // corrisponde alla tabella Genres
}
```

### 2.3.3 Referenziare il database: impostare la stringa di connessione

La classe *context* deve anche stabilire il database con il quale interfacciarsi. Un modo per farlo è impostare la *stringa di connessione* nel metodo `OnConfiguring()`:

```
class Library : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;...");
    }
    public DbSet<Book> Books { get; set; }
    public DbSet<Genre> Genres { get; set; }
}
```

Se la *stringa di connessione* fa riferimento a un database inesistente sarà sollevata un’eccezione.

## 2.4 Mapping fra *entity model* e schema del database

EF esegue un cosiddetto processo di *mapping* per mettere in corrispondenza l’*entity model* con lo schema del database e sollevare degli errori se una corrispondenza precisa non esiste.

Questo processo adotta tre strategie, che possono essere combinate: ***convention***, ***annotation*** e ***fluent api configuration***.

Di seguito introduco la strategia *convention*; le altre due sono introdotte in 5.

## 2.5 *Convention*

EF si affida a delle regole implicite (*convenzioni*) per mettere in corrispondenza lo schema del database con le classi dell’*entity model*. Queste riguardano:

- I nomi dei *dbset* definite nella classe *context*.
- I nomi delle *entity class*.
- I nomi delle proprietà di chiave primaria delle *entity class*.

- I tipi delle proprietà delle *entity class*.
- I nomi delle *proprietà di chiave esterna* delle *entity class*.
- Le *associazioni* esistenti tra le *entity class*. (Le *proprietà di navigazione*.)

Considera nuovamente la classe `Library` e l'*entity model* precedentemente introdotti:

```
class Library : DbContext
{
    ...
    public DbSet<Book> Books { get; set; }         // -> tabella Books
    public DbSet<Genre> Genres { get; set; }       // -> tabella Genres
}

public class Book
{
    public int BookId { get; set; }   // -> chiave primaria BookId
    public string Title { get; set; } // -> colonna Title nvarchar(max)
    public int GenreId { get; set; }  // -> colonna di chiave esterna GenreId
    public Genre Genre { get; set; }  // -> proprietà di navigazione
}                                     //    associazione 1:N tra generi e libri

public class Genre
{
    public int GenreId { get; set; }  // -> chiave primaria GenreId
    public string Name { get; set; }  // -> colonna Name nvarchar(max)
}
```

Sulla base dei due *dbset* e dell'*entity model*, EF deduce l'esistenza delle tabelle **Books** e **Genres**, che definiscono le chiavi primarie **BookId** e **GenreId**. Deduce inoltre l'esistenza delle colonne di testo **Title** e **Name**.

Questa corrispondenza viene creata sulla base delle seguenti convenzioni:

1. Le tabelle hanno il nome delle proprietà `DbSet<>`.
2. Alle proprietà delle *entity class* corrispondono colonne con lo stesso nome.
3. Alle proprietà di nome `<entity>Id` o `Id` corrispondono le colonne di chiave primaria.
4. Se una classe definisce una *proprietà di navigazione* e un campo `<altra-entity>Id`, questo viene considerato una chiave esterna.
5. Ai campi stringa corrispondono colonne di tipo **nvarchar(max)**.

Se, nel tentativo di far corrispondere l'*entity* model allo schema del database, queste convenzioni non sono applicabili, viene sollevata un'eccezione[3].

Ad esempio, supponi che la tabella **Genres** definisca la colonna di chiave primaria **IdGenre**. In questo caso nella classe `Genre` dovremmo definire la proprietà `IdGenre`. Purtroppo EF non riconosce la proprietà `IdGenre` come corrispondente a una chiave primaria, poiché non rispetta il pattern: `<entityclass>Id`. (Vedi: 5.2)

3 In realtà esistono alcuni scenari nei quali anche se l'*entity model* non corrisponde allo schema del database non vengono sollevati errori.

## 2.6 Entità ed entità associate

Una funzione fondamentale di EF è quella di far corrispondere le associazioni tra le classi dell'*entity model* alle associazioni tra le tabelle del database. Ciò consente, quando si carica i record di una tabella, di caricare anche i dati delle tabelle associate.

Ad esempio, nel caricamento dei libri è possibile caricare anche il genere letterario di ogni libro e memorizzarlo nella proprietà `Genre`. È compito di EF eseguire una JOIN tra le tabelle **Genres** e **Books** per ottenere i dati richiesti.

## 2.7 Utilizzo dell'oggetto context

Si crea un'istanza della classe e si usano le proprietà *dbset* per eseguire un'operazione sul database. Il seguente codice visualizza l'elenco dei libri utilizzando *dbset* `Books`:

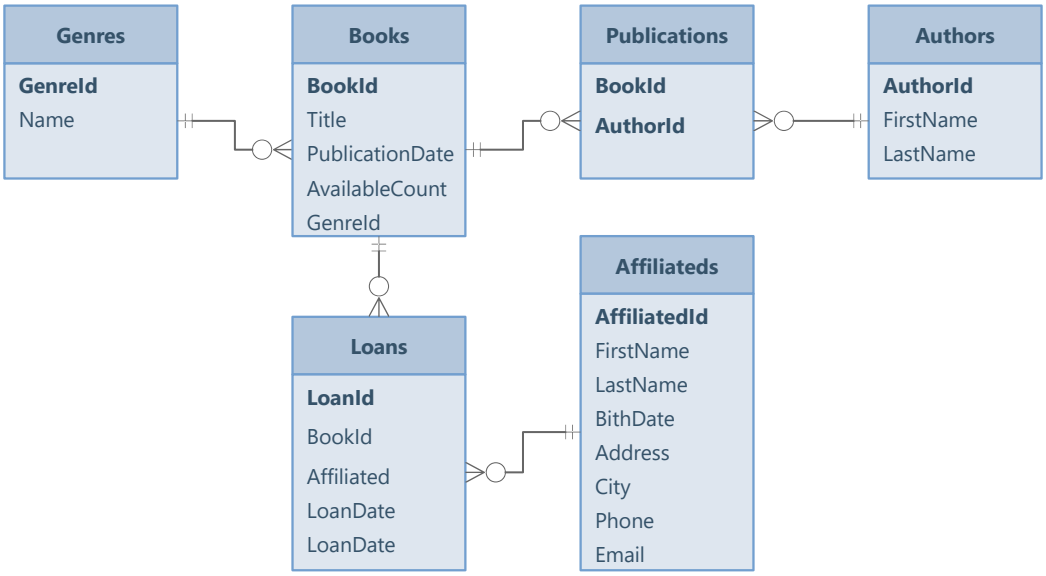
```
var db = new Library();
foreach (var book in db.Books)
{
    Console.WriteLine(book.Title);
}
```

## 2.8 Entity Framework e LINQ

LINQ (***L**anguage **IN**tegrated **Q**uery*) ha un ruolo centrale nell'uso di EF. Ogni interrogazione verso il database viene eseguita con questo linguaggio; sarà EF a tradurre le query LINQ in linguaggio SQL prima di inviarle al database.

# 3 Interrogare il database con Entity Framework

Di seguito affronto alcuni semplici scenari di programmazione allo scopo di mostrare le funzioni principali di EF. Negli esempi sarà usato il database **Library**, che ha il seguente schema:




## 3.1 Definizione dell'*entity model*

Userò inizialmente un *entity model* minimale, ampliandolo col procedere degli esempi:

```
using Microsoft.EntityFrameworkCore;
...
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public DateTime PublicationDate { get; set; }
}

class Library : DbContext
{
    ...
    public DbSet<Book> Books { get; set; }         // -> tabella Books
}
```

## 3.2 Eseguire delle query: LINQ

Una query LINQ inizia specificando un *dbset*. L'esempio seguente ottiene i titoli dei libri:

```
var db = new Library();
var titleList = from book in db.Books
                select book.Title;
```

```
var titleList = db.Books.Select(b => b.Title);
```

```
foreach (var title in titleList)
{
    Console.WriteLine(title);
}
```

La query si legge così: per ogni libro presente in **Books** seleziona il titolo. Il risultato è un elenco di di stringhe. Per la sua esecuzione, EF traduce la query in SQL:

```
SELECT Title FROM Book
```

Il *result set* restituito viene reso accessibile attraverso un oggetto di tipo `IQueryable<string>`.

> **Istruzione SQL generata da EF**
>
> Il codice SQL generato è diverso, anche se identico nel risultato che produce:
>
> `SELECT [b].[Title] FROM [Books] AS [b]`

### 3.2.1 "Esecuzione differita" delle query

Le query vengono eseguite soltanto quando si accede al risultato. Nell'esempio precedente:

```
var db = new Library();
var titleList = from book in db.Books    // <- qui la query NON viene eseguita!
                select book.Title;

foreach (var title in titleList)         // <- la query viene eseguita qui!
{
    Console.WriteLine(title);
}
```

l'istruzione SQL corrispondente alla query viene eseguita soltanto quando si accede alla variabile `titleList`.

### 3.2.2 "Materializzazione" del risultato di una query

Questo comportamento dipende dai tipi `IEnumerable<>` e `IQueryable<>` restituiti dagli operatori LINQ. Il risultato diventa disponibile soltanto quando si "consuma" la query, mediante un `foreach` o un metodo che usa `foreach`, come `ToList()` o `Count()`:

```
var titleList = from book in db.Books
                select book.Title;

int titleCount = titleList.Count();  // <- la query viene eseguita qui!
Console.WriteLine($"Numero libri: {titleCount}");
```

## 3.3 Filtrare e ordinare i dati

Queste operazioni richiedono l'applicazione degli operatori LINQ *where* e *orderby*. Il codice seguente visualizza i libri pubblicati dopo il `1/1/1998`, in ordine di data di pubblicazione:

```
var db = new Library();

DateTime data = DateTime.Parse("1/1/1998");
var bookList = from book in db.Books
               where  book.PublicationDate > data
               orderby book.PublicationDate
               select book;

foreach (var b in bookList)
{
    Console.WriteLine($"{b.Title,-35} {b.PublicationDate:d}");
}
```

Nota bene: EF non è in grado di tradurre in SQL il metodo `DateTime.Parse()`; dunque è necessario assegnare la data a una variabile e usare quest'ultima nella query.

### 3.3.1 Scrivere le query mediante la LINQ API

Una query precedente può essere scritta in modo più conciso utilizzando la API di LINQ:

```
DateTime data = DateTime.Parse("1/1/1998");
var bookList = db.Books.Where(b => b.PublicationDate > data)
                       .OrderBy(b => b.PublicationDate);
```

## 3.4 Caricare un'entità data la chiave primaria

In alcuni scenari si vuole ottenere la singola *entità* che corrisponde a un valore di chiave primaria. Una soluzione consiste nel filtrare il *dbset* alla ricerca dell'entità con la chiave specificata. Il codice seguente trova il libro di chiave 3:

```
var book = db.Books.Where(b => b.BookId == 3).Single();
```

Nota bene: l'uso di `Single()` è necessario per ottenere l'unico oggetto della raccolta restituita da `Where()`.

Esiste in realtà un modo più semplice: utilizzare il metodo `Find()` della classe `DbSet<>`:

```
var book = db.Books.Find(3);
```

# 4 Proprietà di navigazione: usare le associazioni

Una caratteristica fondamentale di EF è la capacità di utilizzare le relazioni tra le *entity class* eseguendo il codice SQL necessario a sfruttare le associazioni delle corrispondenti tabelle.

All'interno dell'*entity model*, le associazioni vengono stabilite mediante le ***proprietà di navigazione***.

## 4.1 Mappare le associazioni: reference property e collection property

EF stabilisce il tipo di associazione tra due *entity class* analizzando le *proprietà di navigazione*, cioè le proprietà che da una classe ne referenziano un'altra.

Considera il seguente modello, contenente due *proprietà di navigazione*:

```
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public Genre Genre { get; set; }            // Reference property
}

public class Genre
{
    public int GenreId { get; set; }
    public string Name { get; set; }
    public List<Book> Books { get; set; }       // Collection property
}
```

La proprietà `Genre` della classe `Book` è una ***reference property***, poiché referenzia il genere di un libro. La proprietà `Books` nella classe `Genre` è una ***collection property***, poiché referenzia i libri appartenenti a un genere. Su queste basi, EF è in grado di stabilire che tra genere e libro esiste un'associazione 1↔N. (Sarebbe in grado di farlo anche se fosse definita soltanto la *reference property* `Genre`.)

Gli altri tipi di associazione sono:

1. Le entità definiscono reciprocamente una *reference property*: EF desume una associazione 1↔1.
2. Le entità definiscono reciprocamente una *collection property*: EF desume una associazione N↔N.

Di seguito introduco l'associazione 1↔N, che di norma non richiede alcuna configurazione. Le altre due sono trattate in (5.6) e (5.7).

## 4.2 Associazione 1↔N

Per mappare una *reference property*, EF deve dedurre il nome della colonna di chiave esterna nella tabella figlia dell'associazione. Per convenzione adotta una delle le seguenti regole:

1. Se la proprietà ha lo stesso nome dell'*entity class* referenziata (il caso comune), EF stabilisce che la colonna di chiave esterna debba chiamarsi: **<nome classe>Id**.
2. Se la proprietà ha un nome diverso, EF stabilisce che la colonna di chiave esterna debba chiamarsi: **<nome proprietà><nome classe>Id**.

Dato l'*entity model* precedente:

```
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public Genre Genre { get; set; }    // Reference property
}                                       // EF deduce la colonna GenreId nella tabella
                                        // Books
public class Genre
{
    public int GenreId { get; set; }
    public string Name { get; set; }
    public List<Book> Books { get; set; }
}
```

EF deduce un'associazione 1↔N tra `Book` e `Genre` e deduce, nella tabella **Books**, l'esistenza di una colonna di chiave esterna di nome **GenreId**. (Il caso comune).

Se la proprietà si chiamasse `Genere`, ad esempio, EF dedurrebbe una colonna di chiave esterna di nome: **GenereGenreId**.

### 4.2.1 Independent associations vs foreign key associations

È possibile definire esplicitamente una proprietà che mappi la colonna di chiave esterna:

```
public class Book
{
    ...
    public int GenreId { get; set; } // EF la interpreta come colonna di chiave esterna
    public Genre Genre { get; set; } // Reference property
}
```

Le associazioni che non definiscono una proprietà di chiave esterna sono definite ***independent association***, mentre quelle che la definiscono si chiamano ***foreign key association***.

Entrambe le scelte sono legittime; d'altra parte, alcuni scenari sono più semplici da gestire se si utilizza una *foreign key association*.

In conclusione: è consigliabile definire le proprietà corrispondenti alle colonne di chiave esterna.

## 4.3 Usare le proprietà di navigazione

Le *proprietà di navigazione* consentono di accedere ai dati delle *entità* correlate. Ad esempio, mediante la *proprietà di navigazione* `Genre` della classe `Book` è possibile accedere al genere letterario di un libro. Di default, però, EF non carica i dati nelle *proprietà di navigazione*, le quali restano inizializzate a `null`.

Ad esempio, considera l'ipotesi di visualizzare l'elenco dei libri e dei rispettivi generi:

```
var db = new Library();
foreach (var b in db.Books)
{
    Console.WriteLine($"{b.Title, -30} {b.Genre.Name}"); // -> NullReferenceException!
}
```

Il codice produce un'eccezione, poiché `b.Genre` è `null`. Infatti, EF si limita a caricare i dati dalla tabella **Books**, traducendo il precedente codice nell'istruzione SQL:

```
SELECT BookId, GenreId, Title FROM Books
```

Per caricare i dati nelle *proprietà di navigazione*, EF consente di adottare tre soluzioni diverse, ognuna delle quali adatta a un particolare scenario: ***eager loading***, ***explicit loading***, ***lazy loading***.

> ### Uso di entità associate già caricate in precedenza
>
> EF è in grado di utilizzare le entità associate se queste sono già state caricate in precedenza dal database.
>
> Considera nuovamente il precedente codice, supponendo però che i generi siano già stati caricati:
>
> ```
> var db = new Library();
> var genres = db.Genres.ToList();    // Carica i generi
> foreach (var b in db.Books)
> {
>     Console.WriteLine($"{b.Title, -30} {b.Genre.Name}"); // OK
> }
> ```
>
> In questo caso, EF assegna alla proprietà `Genre` di ogni libro il genere corrispondente già in memoria.

## 4.4 Eager loading

La strategia *eager loading* implica l'immediato caricamento dei dati richiesti, eseguendo una *join* tra le tabelle corrispondenti alle *entity class*.

Si applica l'*eager loading* mediante il metodo `Include()`, il quale accetta una *lambda expression* che specifica l'entità associata da includere nella query.

Ad esempio, il codice che carica i libri e i relativi generi può essere scritto nel seguente modo:

```
var db = new Library();
foreach (var book in db.Books.Include(b => b.Genre))      // carica anche generi
{
    Console.WriteLine($"{b.Title, -30} {b.Genre.Name}");  // funziona correttamente
}
```

L'uso del metodo `Include()` viene tradotto in una JOIN tra **Books** e **Genres**:

```
SELECT BookId, GenreId, Title, Genres.GenreId, Genres.Name
FROM Books
INNER JOIN Genres ON Books.GenreId = Genres.GenreId
```

L'*eager loading* funziona anche con le *collection property*, come mostra il seguente codice, che carica i generi e, per ognuno di essi, i libri che vi appartengono:

```
var db = new Library();
foreach (var g in db.Genres.Include(g => g.Books))
{
    Console.WriteLine($"{g.Name}");
    foreach (var b in g.Books)              // scorre i libri del genere
    {
        Console.WriteLine($"\t{b.Title}");
    }
}
```

## 4.5 Explicit loading

L'*eager loading* si applica alle query che restituiscono una raccolta, ma esistono scenari nei quali occorre caricare i dati correlati di una specifica *entità*.

Ad esempio, un'applicazione di gestione della biblioteca prevede la visualizzazioni dei titoli dei libri e la possibilità di selezionare un libro per visualizzarne i dettagli, dunque anche il genere letterario. In questo scenario non è efficiente applicare l'*eager loading*, poiché sarebbero caricati in anticipo i generi di tutti i libri.

In questi casi è utile l'*explicit loading*. Una volta caricata l'*entità* (il libro), si usa il metodo `Entry()`, il quale restituisce un oggetto di tipo `EntityEntry`, che espone i metodi `Reference()` e `Collection()`, utilizzabili per caricare le *entità* associate.

Il seguente codice carica il primo libro dal database e, di seguito, il relativo genere.

```
var db = new Library();
Book book = db.Books.First();           // carica il primo libro, senza caricare il genere
...
var entry = db.Entry(book);             // ottiene l'oggetto entry associato al libro
entry.Reference(b => b.Genre).Load();   // il genere viene caricato qui
Console.WriteLine(book.Genre.Name);
```

Il precedente codice produce due query SQL. La prima carica i dati del libro:

```
SELECT TOP(1) BookId, GenreId, Title FROM Books
```

La seconda, corrispondente all'esecuzione del metodo `Load()`, carica i dati del genere correlato:

```
SELECT GenreId, Name FROM Genres WHERE GenreId = 1  -- genere Fantascienza
```

## 4.6 Lazy loading

Il termine *lazy loading* – letteralmente: "caricamento pigro" – designa la capacità di caricare automaticamente i dati, ma soltanto quando sono richiesti. Svolge una funzione analoga all'*explicit loading*, ma senza che il programmatore debba scrivere il codice necessario per caricare le entità correlate.

L'uso del *lazy loading* richiede l'istallazione di una libreria e la configurazione della classe *context* (vedi Appendice II: istallare e configurare il lazy loading). Inoltre, richiede che tutte le proprietà di navigazione siano virtuali:

```
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public int GenreId { get; set; }
    public virtual Genre Genre { get; set; }
}

public class Genre
{
    public int GenreId { get; set; }
    public string Name { get; set; }
    public virtual List<Book> Books { get; set; }
}
```

L'applicazione del *lazy loading* semplifica notevolmente il codice precedente:

```
var db = new Library();
var book = db.Books.First();              // qui carica il libro
Console.WriteLine(book.Genre.Name);       // qui carica il genere
```

### 4.6.1 Problematiche sull'uso del lazy loading

Il *lazy loading* appare la strategia migliore, perché è semplice ed evita il caricamento anticipato di dati che potrebbero rivelarsi non necessari. Presenta comunque delle problematiche.

Innanzitutto deve essere configurato. Secondo, richiede che tutte le proprietà di navigazione siano dichiarate virtuali, anche quelle che non vengono utilizzate con questa strategia. Infine, nasconde potenziali problemi di performance.

Considera il seguente codice, che visualizza i libri e i relativi generi:

```
var db = new Library();
foreach (var book in db.Books)      // query dei soli libri
{
```

```
    Console.WriteLine("{0} {1}", book.Title, book.Genre.Name); // query del genere del libro
}
```

Il caricamento dei libri nella prima query non produce il caricamento dei generi correlati. Nel ciclo, l'accesso al genere di ogni libro determina l'esecuzione di una query per il suo caricamento. In conclusione, per la visualizzazione di 100 libri verrebbero eseguite 101 query!

> ### Lazy loading e *caching* delle entità correlate
>
> In realtà il *lazy loading* è più efficiente di quanto descritto nell'esempio precedente. Le entità correlate, una volta caricate in memoria, vengono riutilizzate nel caso vi sia una successiva richiesta.
>
> Ad esempio, supponi che vi siano 100 libri disponibili, ma appartenenti a cinque generi soltanto. In questo caso, il codice precedente produrrebbe una query per i libri e una per ognuno dei generi. Infatti, prima di caricare un genere dal database, EF verifica che non sia già in memoria.

# 5 Configurazione dell'*entity model*

Negli scenari più semplici le convenzioni adottate da EF sono sufficienti per "mappare" l'*entity model* con lo schema del database. L'*entity model* implementato nel capitolo precedente ricade in questa situazione:

1. Le chiavi primarie hanno il nome `<entità>Id` (o `Id`).
2. Non esistono associazioni 1↔1 o N↔N (ma solo 1↔N).
3. Le chiavi esterne sono campi interi e hanno il nome: `<proprietà>Id`.
4. Ad ogni tabella corrisponde un *dbset* nella classe *context*.
5. Non sono definiti vincoli sulle proprietà delle *entity class*.

In altri scenari, però, è necessario andare oltre le convenzioni e applicare delle configurazioni manuali, in modo che EF riesca a mappare correttamente l'*entiry model*.

Esistono due strategie, che possono essere impiegate contemporaneamente:

- ***annotation***: le annotazioni sono degli attributi che decorano agli elementi (classi e proprietà) dell'*entity model*. Sono definiti nei *namespace*:

  `System.ComponentModel.DataAnnotation`

  `System.ComponentModel.DataAnnotation.Schema`

- ***fluent API configuration***: dei metodi che consentono di configurare l'*entity model*.

Di seguito introdurrò alcune di queste configurazioni, contestualmente all'introduzione di nuove *entity class* corrispondenti allo schema definito in 3.

## 5.1 Mapping di tabelle: attributo [Table]

Di default EF mappa i nomi delle tabelle del database con i nomi dei *dbset* della classe *context*. Potrebbe accadere, però, di voler usare una nomenclatura diversa rispetto al database (italiano vs inglese, ad esempio). Oppure di non voler definire un *dbset* per ogni tabella.

Considera il secondo scenario e ipotizza di definire il solo *dbset* `Books`:

```
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public int GenreId { get; set; }
    public Genre Genre { get; set; }
}

public class Genre    // -> in assenza di dbset, corrisponde alla tabella Genre
{
    public int GenreId { get; set; }
    public string Name { get; set; }
}
```

```
class Library : DbContext
{
    ...
    public DbSet<Book> Books { get; set; }         // -> tabella Books
}
```

In assenza di *dbset*, l'*entity class* `Genre` viene mappata con la tabella **Genre**, che però non esiste. Occorre stabilire mappare esplicitamente la tabella mediante l'attributo `[Table]`.

```
using System.ComponentModel.DataAnnotations.Schema;

[Table("Genres")]
public class Genre
{
    public int GenreId { get; set; }
    ...
}
```

Nota bene: l'uso dell'attributo può convivere con la definizione del *dbset* corrispondente alla tabella. Se vengono utilizzati entrambi, EF darà precedenza all'attributo.

## 5.2 Mappare la chiave primaria: attributo [Key]

Di default, EF interpreta una proprietà di nome `Id` o `<entityclass>Id` come corrispondente alla chiave primaria della tabella. Ma la chiave primaria potrebbe chiamarsi diversamente; in questo caso è possibile utilizzare l'attributo `[Key]`.

Ad esempio, se la tabella **Books** definisse la colonna ISBN come chiave primaria, la classe `Book` dovrebbe essere configurata così:

```
using System.ComponentModel.DataAnnotations;

public class Book
{
    [Key]
    public string ISBN { get; set; }
    public string Title { get; set; }
    ...
}
```

## 5.3 Mappare le colonne: attributo [Column]

Con l'attributo `[Column]` è possibile personalizzare il mapping con le colonne del database, ad esempio specificando il nome e il tipo della colonna corrispondente alla proprietà.

Supponi che il titolo dei libri sia definito dalla colonna **BookTitle**, ma di voler comunque usare il nome `Title` per la proprietà corrispondente:

```
using System.ComponentModel.DataAnnotations.Schema;
```

```
public class Book
{
    public int BookId { get; set; }

    [Column("BookTitle")]
    public string Title { get; set; }

    public int GenreId { get; set; }
    public Genre Genre { get; set; }
}
```

## 5.4 Evitare il mapping di una proprietà

EF mappa tutte le proprietà scrivibili di un'*entity class* e per esse "pretende" che nel database esistano le colonne corrispondenti. Eventuali proprietà derivate o *read-only* non vengono mappate, dunque è legittimo scrivere il seguente codice anche se nella tabella **Authors** non esiste la colonna **FullName**:

```
public class Author
{
    public int AuthorId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public string FullName => LastName + ", " + FirstName;   // non viene mappata
}
```

In alcuni scenari, però, potrebbe essere utile definire delle proprietà scrivibili che non abbiano una corrispondenza nel database. L'attributo `[NotMapped]` risolve questo problema, evitando che una proprietà venga mappata.

Il codice seguente aggiunge la proprietà `FullName` ad `Author`, ma la decora con `[NotMapped]` e dunque evita che venga mappata.

```
public class Author
{
    public int AuthorId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string FullName => LastName + ", " + FirstName;

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

### 5.4.1 Usare le colonne non mappate nelle query

Poiché non corrispondono a colonne nel database, le proprietà *not mapped* non possono essere usati nelle query; infatti, la traduzione della query in SQL fa riferimento a colonne esistenti nel database.

Ad esempio, la seguente operazione produce un errore di esecuzione:

```
var authors = db.Authors.Where(a => a.FullName.StartsWith("A"));
foreach (var a in authors)    // -> l'eccezione viene prodotta qui!
{
    Console.WriteLine(a.FullName);
}
```

È possibile filtrare su `FullName` se la query viene eseguita in memoria, invocando prima `ToList()` e soltanto dopo applicando il filtro

```
var authors = db.Authors.ToList().Where(a => a.FullName.StartsWith("As"));
```

## 5.5 Configurare le associazioni 1↔N

Di norma non c'è bisogno di configurare le associazioni 1↔N, salvo definire una proprietà corrispondente alla colonna di chiave esterna (4.2.1). Esistono scenari, comunque, nei quali occorre stabilire esplicitamente quale proprietà corrisponde alla colonna di chiave esterna. In questi casi si può usare l'attributo `[ForeignKey]`.

Di seguito ipotizzo che la chiave esterna che associa **Books** a **Genres** si chiami **FK_Genre**:

```
public class Book
{
    ...

    public int FK_Genre { get; set; }      // corrisponde alla colonna di chiave esterna

    [ForeignKey("FK_Genre")]               // indica a EF la proprietà che mappa la FK
    public Genre Genre { get; set; }

}
```

## 5.6 Configurare le associazioni N↔N: usare la fluent API

EF riconosce un'associazione N↔N quando due *entity class* si referenziano reciprocamente mediante *collection property*. EF mappa l'associazione con una tabella di collegamento (*join table*) contenente le chiavi delle due entità, le quali formano la chiave primaria composta della tabella:

**Entity model**

```
public class Book
{
    ...
    public List<Author> Authors { get; set; }
}
public class Author
{
    ...
    public List<Book> Books { get; set; }
}
```

**Database**

Il nome della tabella rispetta il pattern **<classe1><classe2>**[4]. Il nome delle chiavi rispetta il pattern:

**<Nome proprietà><chiave primaria>**.

Se si desidera modificare il mapping è necessario utilizzare la Fluent API, poiché in questo caso, non esistendo un'*entity class* corrispondente alla tabella di collegamento, non è possibile usare le annotazioni.

### 5.6.1 Uso della Fluent API per configurare le relazioni N↔N

Nel database **Library**, la *join table* tra libri ed autori si chiama **Publications**. Inoltre le colonne di chiave esterna rispettano il pattern **<nome entità>Id**.

Nel metodo virtuale `OnModelCreating()` della classe *context* si usa l'oggetto `ModelBuilder` per configurare ogni elemento dell'associazione:

```
public class Library : DbContext
{
    ...
    protected override void OnModelCreating(ModelBuilder mb)
    {
        mb.Entity<Author>()             // un Author
          .HasMany(a => a.Books)        // ha molti Books
          .WithMany(b => b.Authors)     // ognuno dei quali ha molti Author

          .UsingEntity<Dictionary<string, object>>("Publications",
              x => x.HasOne<Book>().WithMany().HasForeignKey("BookId"),
              y => y.HasOne<Author>().WithMany().HasForeignKey("AuthorId"));
    }
}
```

Si tratta di un'unica istruzione suddivisa in due parti. Si configura innanzitutto l'associazione nell'*entity model*, stabilendo che un autore ha 'n' libri e un libro ha 'n' autori. Dopodiché, mediante il metodo `UsingEntity()`, si definiscono la tabella di collegamento e le colonne di chiave esterna usate per implementare l'associazione nel database.

### 5.6.2 Associazioni N↔N con attributi propri

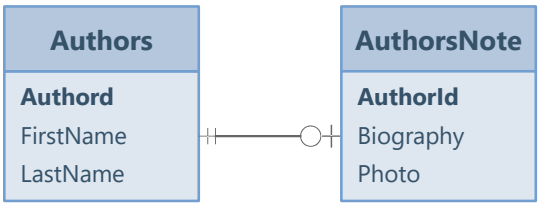
Le associazioni N↔N che definiscono anche degli attributi non possono essere mappate automaticamente da EF come descritto nel precedente paragrafo. In questo caso è necessario trattare l'associazione come una normale tabella e definire un'*entity class* che la mappi.

È il caso dell'*entity class* `Loan`, che presenta un'associazione N↔1 sia verso `Affiliated` che verso `Book`. EF non fa alcuna distinzione tra una normale associazione 1↔N e una che è parte di un'associazione N↔N. Quindi, per entrambe valgono le stesse regole di mapping.

4 Nello scegliere quale classe mettere per prima nel nome, EF usa l'ordine alfabetico.

## 5.7 Configurare associazioni 1↔1

Nel database **Library** esiste un'associazione 1↔1 tra le tabelle **Authors** e **AuthorsNote**:




Nella tabella **AuthorsNote**, la colonna **AuthorId** funge sia da chiave primaria (non *identity*), sia da chiave esterna. Nel modello è necessario aggiungere la classi `AuthorNote` e modificare la classe `Author`. Nella classe `AuthorNote` è necessario usare l'attributo `[Table]` per indicare il nome della tabella corrispondente.

```
public class Author
{
    public int AuthorId { get; set; }
    ...
    public AuthorNote AuthorNote { get; set; }    // ref. property verso AuthorNote
}

[Table("AuthorsNote")]
public class AuthorNote
{
    public int AuthorId { get; set; }        // viene considerata chiave esterna
    public byte[] Photo { get; set; }
    public string Biography { get; set; }
    public Author Author { get; set; }       // ref. property verso Author
}
```

Tutto ciò non è sufficiente, perché EF interpreta `AuthorId` di `AuthorNote` come proprietà di chiave esterna, ma non come proprietà di chiave primaria. Occorre annotare il campo `AuthorId` con l'attributo `[Key]` indicando che svolge entrambi i ruoli:

```
[Table("AuthorsNote")]
public class AuthorNote
{
    [Key]
    public int AuthorId { get; set; }      // è chiave primaria e chiave esterna
    public Author Author { get; set; }

    public byte[] Photo { get; set; }
    public string Caption { get; set; }
}
```

## 5.8 Valori obbligatori / facoltativi

Nel modello relazionale le colonne di una tabella possono essere richieste (NOT NULL), oppure no, indipendentemente dal tipo della colonna. In C# la "nullabilità" di una variabile/proprietà dipende dal suo tipo.

Nel creare una corrispondenza tra l'*entity model* e lo schema del database, EF segue le regole del linguaggio C# per stabilire se un valore è richiesto oppure no. Precisamente: i *value type* (`int`, `double`, `DateTime`, etc) sono considerati richiesti, mentre i *reference type* (le classi) sono considerati non richiesti. In sintesi, EF considera NOT NULL le colonne corrispondenti alle proprietà *value type*.

A questo riguardo, EF tollera un'incongruenza tra l'*entity model* e lo schema del database, salvo produrre un'eccezione oppure dei dati errati se vengono eseguite operazioni che violano un vincolo sul valore delle colonne o delle proprietà.

Di seguito mostro come eliminare queste incongruenze in modo che l'*entity model* e il database siano allineati.

## 5.9 Rendere opzionali i "tipi valore"

La tabella **Loans** memorizza le informazioni sui prestiti dei libri e tra le sue colonne ci sono la data di prestito e quella di restituzione. La prima è una colonna richiesta, mentre **ReturnDate** è non richiesta, poiché il valore viene inserito soltanto all'atto della restituzione del libro.

```
CREATE TABLE Loans (
    LoanId       INT      IDENTITY (1, 1) NOT NULL,
    BookId       INT      NOT NULL,
    AffiliatedId INT      NOT NULL,
    LoanDate     DATETIME NOT NULL,    -- data di prestito (richiesta)
    ReturnDate   DATETIME NULL,        -- data di restituzione (non richiesta)
...
)
```

Nell'*entity model*, il tipo `DateTime` non ammette valori `null` e ciò può comportare dei problemi sia nel caricamento dei dati che nella loro modifica.

```
public class Loan
{
    public int LoanId { get; set; }
    public int BookId { get; set; }
    public int AffiliatedId { get; set; }
    public DateTime LoanDate { get; set; }   // data di prestito (richiesta)
    public DateTime ReturnDate { get; set; } // data di restituzione (richiesta)
}
```

Il seguente codice è destinato a fallire nel momento in cui viene caricata una riga della tabella **Loans** che contiene NULL in **ReturnDate**:[5]

```
var db = new Library();
```

5 In scenari simili non sempre viene sollevata un'eccezione; in base al tipo di query che viene eseguita potrebbe accadere che nella proprietà `ReturnDate` venga memorizzato il valore predefinito, 1/1/0001.

```
foreach (var loan in db.Loans)  // qui si verifica l'eccezione SqlNullValueException
{
    Console.WriteLine($"{loan.LoanDate} {loan.ReturnDate}");
}
```

La soluzione è rendere la proprietà `ReturnDate` *nullabile*, utilizzando il tipo `DateTime?` (*nullable value type*):

```
public class Loan
{
    public int LoanId { get; set; }
    public int AffiliatedId { get; set; }
    public int BookId { get; set; }
    public DateTime LoanDate { get; set; }
    public DateTime? ReturnDate { get; set; }
}
```

Dopo questa modifica, la proprietà `ReturnDate` può memorizzare valori nulli, coerentemente con la colonna **ReturnDate**.

## 5.10 Rendere obbligatorie le proprietà nullabili

In questo caso siamo davanti al problema opposto: imporre un valore obbligatorio in proprietà che, per loro natura, sono *nullabili* (tipicamente, proprietà stringa).

Si tratta di una questione complessa, poiché dipende da una funzione del linguaggio chiamata ***N**ullable **R**eference **T**ypes;* funzione che può essere abilitata o meno (nei progetti creati con .NET 6 è attiva di default). Di seguito ignoro questa funzione (la considero disabilitata), che affronto in Appendice IV: Proprietà nullabili e NRT.

L'attributo `[Required]` applicato a proprietà *nullabili* impone che abbiano un valore durante le operazioni da/verso il database. Ad esempio, il seguente codice impone la *non-nullabilità* del nome e del cognome degli autori:

```
public class Author
{
    public int AuthorId { get; set; }

    [Required]
    public string FirstName { get; set; }

    [Required]
    public string LastName { get; set; }
    ...
}
```

Ciò ha due implicazioni. La prima riguarda lo scenario di generazione automatica del database a partire dall'*entity model*, tema che tratto in Appendice V: generare il database dal modello).

La seconda riguarda la possibilità che, nel database, la colonna corrispondente alla proprietà sia *nullabile*. In questo caso il caricamento di un record privo di valore nella colonna produce un errore, poiché EF si "rifiuta" di assegnare `null` alla proprietà.

Nella pratica, eccettuato lo scenario descritto in Appendice V: generare il database dal modello, l'attributo `[Required]` è superfluo. Il perché è abbastanza semplice da comprendere:

- Se la colonna corrispondente alla proprietà è NOT NULL, non accadrà mai di caricare un record con un valore nullo in quella colonna, dunque non c'è alcuna necessità di imporre un vincolo sulla proprietà.
- Se la colonna corrispondente alla proprietà è *nullabile*, significa che anche la proprietà deve essere *nullabile*, oppure che c'è un problema di progettazione. (E imporre `[Required]` sulla proprietà non risolve il problema.)

> ### Proprietà obbligatorie e TPH Strategy
>
> Nell'implementazione di associazioni di generalizzazione, precisamente con l'adozione della strategia TPH descritta In 8.1, l'uso di `[Required]` è utile, poiché nel database è obbligatorio usare colonne *nullabili* anche per quei dati che dovrebbero essere obbligatori.

# 6 Modifica dei dati

EF consente di aggiungere, eliminare, modificare entità, e successivamente di salvare sul database le modifiche mediante la semplice chiamata a un metodo. Le funzioni fornite sono:

1. Tracciare le modifiche alle *entità*, la loro eliminazione e il loro inserimento (traducendo le modifiche in operazioni INSERT, DELETE o UPDATE).
2. Inviare le modifiche tenendo conto delle associazioni che esistono tra le *entità*.
3. Recuperare automaticamente la chiave primaria dopo l'inserimento di una nuova *entità*.
4. Validare le modifiche prima che vengano inviate al database.
5. Gestire la concorrenza.

## 6.1 Gestione in memoria delle entità: *change tracking*

Perché EF sia in grado di gestire correttamente le operazioni di aggiornamento è necessario che tenga traccia delle modifiche alle *entità* coinvolte, in modo da poter tradurre correttamente tali modifiche nelle corrispondenti istruzioni INSERT, UPDATE, DELETE.

Considera il seguente codice, che crea due liste di generi, la prima in memoria, la seconda ottenuta dal database:

```
var lista1 = new List<Genre>
{
    new  Genre {GenreId = 1,  Name = "Fantascienza"},
    new  Genre {GenreId = 2,  Name = "Fantasy"},
    new  Genre {GenreId = 3,  Name = "Horror"}
};

var db = new Library();
var lista2 = db.Genres.Take(3).ToList();  // carica i primi tre generi dal database
```

I generi memorizzati in `lista1` sono semplici oggetti creati in memoria; quelli memorizzati di `lista2` rappresentano una copia esatta dei record della tabella **Genres**. Una modifica a quest'ultimi viene rilevata da EF, il quale è in grado di eseguire le istruzioni SQL appropriate in caso di modifica delle *entità*.

Questo è possibile perché EF memorizza internamente lo stato di ogni *entità* caricata dal database, tenendo traccia delle eventuali modifiche che subisce.

### 6.1.1 Stato di un'entità: proprietà State

A seguito di un'operazione, un'*entità* può trovarsi in uno dei seguenti stati:

| Stato | Descrizione |
|---|---|
| **Added** | L'*entità* è stata inserita nel *context* e non esiste ancora nel database. |
| **Deleted** | L'*entità* è considerata eliminata nel *context*, ma esiste ancora nel database. |

| Stato | Descrizione |
|---|---|
| **Detached** | L'*entità* è stata creata in memoria e non è ancora tracciata dal *context*. (Si chiama ***entità disconnessa***. )<br>Non sarà coinvolta nelle operazioni verso il database. |
| **Modified** | L'*entità* è considerata modificata nel *context*. Nel database esiste ancora la versione originale. |
| **Unchanged** | L'*entità* non ha subito alcuna modifica rispetto a quella esistente nel database. |

Tra le funzioni accessibili con `Entry()` c'è la proprietà `State`, che memorizza lo stato dell'*entità*. Il seguente codice visualizza lo stato del primo genere della tabella (in questo caso *Unchanged*):

```
var db = new Library();
var genre = db.Genres.First();
EntityEntry entry = db.Entry(genre);
Console.WriteLine(entry.State);  // -> Unchanged
```

## 6.2 Salvare le modifiche nel database

Per salvare le modifiche nel database si usa il metodo `SaveChanges()` dell'oggetto *context*:

```
var db = new Library();
//
// inserisce, modifica, elimina una o più entità
//
db.SaveChanges();  // il database viene aggiornato; tutte le entità diventano unchanged
```

### 6.2.1 Esecuzione “transazionale” delle modifiche

Il metodo `SaveChanges()` esegue le modifiche nel database in una ***transazione***: o tutte le modifiche vengono eseguite correttamente, oppure viene eseguito un ***rollback***, cioè un ripristino allo stato precedente l'aggiornamento.

In caso di errore il metodo solleva un’eccezione; è compito dell’applicazione catturarla e gestirla, ad esempio visualizzando un messaggio appropriato.

> **SaveChanges() e pattern “*Unit of Work*”**
>
> Il pattern *Unit of Work* definisce l'abilità di gestire un insieme di modifiche come una singola “unità di lavoro”, che può essere completata o annullata, ma non eseguita parzialmente.

## 6.3 Inserimento di un'entità

Innanzitutto, si aggiunge la nuova *entità* al *dbset* mediante il metodo `Add()`. Il codice seguente inserisce un nuovo genere:

```
var genre = new Genre { Name = "Narrativa" };  //genre è detached
var db = new Library();
db.Genres.Add(genre);                 // genre è added
db.SaveChanges();                     // genre è inserito nel database (unchanged)
Console.WriteLine(genre.GenreId);     // GenreId è stato generato dal database
```

`Add()` aggiunge l'*entità* in memoria: questa non viene salvata nel database fintantoché non viene eseguito il metodo `SaveChanges()`.

### 6.3.1 Recupero della chiave primaria identity

Nel precedente esempio, `SaveChanges()` inserisce il nuovo genere, recupera la chiave primaria generata dal DBMS e la memorizza nel campo `GenreId`.

Il recupero della chiave primaria viene eseguito soltanto se è *identity*; in caso contrario è compito dell'applicazione inserire un valore valido nella *proprietà di chiave primaria* prima di aggiungere la nuova entità.

### 6.3.2 Ripetere l'inserimento in caso di errore

Il tentativo di ripetere l'inserimento di un'*entità* a causa di un precedente errore presenta un potenziale problema nel caso in cui venga usato lo stesso *context*.

Supponi di implementare l'inserimento nel seguente modo:

```
var db = new Library();   // db è globale
...
void btnInserisci_Click(object sender, EventArgs e)
{
    var genre = new Genre { Name = txtNomeGenere.Text };

    db.Genres.Add(genre);  // aggiunge il genere in memoria
    try
    {
        db.SaveChanges();
    }
    catch (Exception ex)
    {
       // -> visualizza un messaggio di errore
    }
}
```

L'utente tenta di inserire il genere **Fantascienza** e riceve un errore, poiché esiste già nel database. Dopodiché inserisce un nuovo nome, questa volta inesistente, ma riceve ugualmente un errore di violazione del vincolo univoco.

Il problema consiste nel fatto che il precedente genere è ancora memorizzato nel *context* con lo stato `Added`. La seconda esecuzione di `SaveChanges()` tenta di inserire entrambi i generi, e dunque produce un errore relativo al primo.

La soluzione consiste nel creare un nuovo *context* per l'inserimento, oppure di rimuovere dal *dbset* il genere che ha prodotto l'errore.

```
...
db.Genres.Add(genre);
try
{
    db.SaveChanges();
}
catch (Exception ex)
{
   db.Remove(genre);
   // visualizza messaggio di errore
}
```

## 6.4 Modificare un'entità

La modifica di un'*entità* si ottiene cambiando una o più proprietà e successivamente invocando `SaveChanges()`. Perché la modifica abbia successo, l'*entità* deve essere tracciata dal *context*.

Il codice seguente modifica la data di pubblicazione del primo libro della tabella:

```
var db = new Library();
var book = db.Books.Find(1);
book.PublicationDate = DateTime.Parse("1/1/2001");    // book è modified
db.SaveChanges();                                      // book è unchanged
```

Nell'inviare la modifica, EF la traduce in un'istruzione UPDATE simile alla seguente:

```
UPDATE Books
    SET PublicationDate = @0
WHERE BookId = @1

@0='1/1/2001'
@1=1
```

## 6.5 Eliminare un'entità

L'eliminazione di un'*entità* si ottiene eseguendo il metodo `Remove()` del *dbset*.

Il codice seguente recupera il genere **Narrativa** e lo rimuove:

```
var db = new Library();
var genre = db.Genres.Where(g => g.Name == "Narrativa").Single(); // carica il genere
db.Genres.Remove(genre);     // genre è deleted
db.SaveChanges();            // il genere è stato eliminato dal database
```

(Nota bene: il metodo `Single()` ritorna l'unico elemento della query.)

### 6.5.1 Eliminare un'entità ancora non persistita sul database

Se un'*entità* è stata appena inserita nel *dbset*, può essere eliminata semplicemente eseguendo `Remove()`. In questo caso `SaveChanges()` non produrrà alcuna modifica al database.

```
Genre newGenre = new Genre { Name = "Narrativa" };
var db = new Library();
db.Genres.Add(newGenre);            // aggiunge l'entità al contesto
db.Genres.Remove(newGenre);         // elimina l'entità dal contesto
db.SaveChanges();                   // inutile: non produce alcuna azione sul database
```

### 6.5.2 Eliminare un'entità senza caricarla dal database

In alcuni scenari l'*entità* da eliminare non è stata caricata in memoria, ma di essa si conosce la chiave primaria. In questo caso è possibile registrarla per l'eliminazione senza doverla prima caricare in memoria.

Si crea un'entità "fittizia" (*stub entity*), contenente soltanto la chiave primaria, quindi la si "attacca" al *context* mediante il metodo `Attach()`, infine la si elimina.

Il codice seguente elimina il genere di chiave 9:

```
var db = new Library();
Genre genre = new Genre { GenreId = 9 };    // crea un'entità fittizia
db.Genres.Attach(genre);                    // attacca l'entità al contesto
db.Genres.Remove(genre);                    // registra l'entità per eliminazione
db.SaveChanges();                           // elimina il genere nel database
```

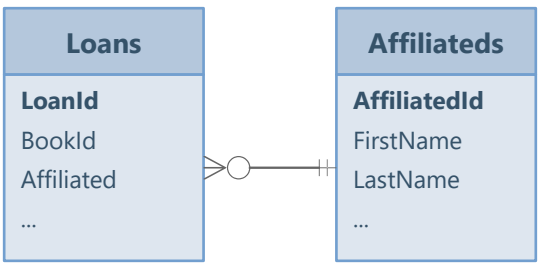
### 6.5.3 Eliminare un'entità "padre" di un'associazione

Questa scenario si presenta in varie configurazioni:

1. Nel database è definita/non definita una *regola di eliminazione a cascata*.
2. Nell'*entity model* (ma non nel database) è definita una *regola di eliminazione a cascata*.
3. L'associazione è opzionale (possono esistere record orfani).
4. Le *entità* figlie dell'associazione sono/non sono caricate in memoria.

Qui prendo in considerazione lo scenario standard, nel quale il database definisce un *vincolo di integrità referenziale*, ma non la *regola di eliminazione a cascata*. In questo caso è responsabilità del codice eliminare prima tutte le entità figlie e soltanto successivamente quella padre.

Considera l'associazione **Affiliateds - Loans** (1→N):



Segue l'*entity model*, con le sole proprietà necessarie a mappare l'associazione:

```
public class Library : DbContext
{
    ...
    public DbSet<Affiliated> Affiliateds { get; set; }
    public DbSet<Loan> Loans { get; set; }
}

public class Affiliated
{
    public int AffiliatedId { get; set; }
    public List<Loan> Loans { get; set; }  // indica a EF il tipo di associazione
    ...
}

public class Loan
{
    public int LoanId { get; set; }
    public int AffiliatedId { get; set; }    // necessario a EF per conoscere la FK
    ...
}
```

Ipotizza di voler eliminare il tesserato di chiave 3:

```
var db = new Library();
var tesserato = db.Affiliateds.Find(3);    // Carica il tesserato di chiave primaria 3
db.Entry(tesserato).Collection(t => t.Loans).Load();  // carica i prestiti
foreach (var prestito in tesserato.Loans)
{
    db.Loans.Remove(prestito);           // rimuove tutti i prestiti del tesserato
}
db.Affiliateds.Remove(tesserato);       // rimuove il tesserato
db.SaveChanges();                       // elimina i prestiti e il tesserato dal database
```

Nell'ordine:

1. Viene caricato il tesserato.
2. Sono caricati i prestiti del tesserato e vengono registrati per l'eliminazione.
3. Il tesserato viene registrato per l'eliminazione.
4. Vengono salvate le modifiche, cioè eliminati i prestiti e il tesserati che li ha ricevuti.

Segue lo script SQL che esegue l'intera operazione, a partire dal caricamento dell'affiliato:

```
-- carica il tesserato

SELECT TOP (1) AffiliatedId FROM Affiliateds WHERE AffiliatedId = @p0
@p0 int, @p0=3
go
```

```
-- carica i prestiti del tesserato
SELECT LoanId, AffiliatedId  FROM Loans WHERE AffiliatedId = @p0
@p0 int,  @p0=3
go

DELETE FROM Loans WHERE LoanId = @p0               -- elimina primo prestito
@p0 int, @p0=6
go

DELETE FROM Loans WHERE LoanId = @p0               -- elimina secondo prestito
@p0 int, @p0=7
go

DELETE FROM Affiliateds WHERE AffiliatedId = @p1   -- elimina tesserato
@p0 int, @p0=3
go
```

## 6.6 Modificare le associazioni

Di seguito saranno considerati alcuni scenari nei quali si rende necessario impostare o modificare l'associazione tra due *entità*. Per impostare/modificare/rimuovere un'associazione è possibile agire sia sulla *collection property* dell'*entità* padre, che sulla *reference property* o proprietà di chiave esterna dell'*entità* figlia.

### 6.6.1 Inserimento di un'entità figlia di un'associazione

Ipotizziamo di voler inserire un nuovo libro, assegnandolo al genere **Fantascienza**. Una soluzione consiste nell'ottenere l'entità del genere **Fantascienza** e aggiungere il libro alla sua *collection property* `Books`:

```
var libro = new Book
{
    Title = "Paratwa",
    PublicationDate = DateTime.Parse("1/1/2003"),
    AvailableCount = 1
};
var db = new Library();
var genere = db.Genres.Where(g => g.Name == "Fantascienza").Single();  // carica il genere
db.Entry(genere).Collection(g => g.Books).Load(); // Carica i libri del genere
genere.Books.Add(libro);    // aggiunge il libro al genere fantascienza
db.SaveChanges();           // inserisce i libro impostando correttamente la sua FK
```

Alternativamente, si può aggiungere il libro al *dbset* `Books` e impostare la sua *reference property* `Genre` al genere precedentemente caricato:

```
var libro = new Book { ... };
var db = new Library();
var genere = db.Genres.Where(g => g.Name == "Fantascienza").Single();  // carica il genere
db.Books.Add(libro);  // aggiunge il libro all'elenco dei libri
libro.Genre = genere;  // imposta il genere del libro (usa la reference property di Book)
db.SaveChanges();
```

(Nota bene, con questa soluzione non è necessario eseguire l'*explicit loading* dei libri mediante il metodo `Entry()`.)

Infine, è possibile impostare direttamente la proprietà di chiave esterna `GenreId` del libro con il valore di chiave primaria del genere di **Fantascienza**:

```
...
db.Books.Add(libro);                // aggiunge il libro all'elenco dei libri
libro.GenreId = genere.GenreId;     // imposta la FK del libro con la PK del genere
db.SaveChanges();
```

In ogni caso, l'esecuzione di `SaveChanges()` produce l'inserimento del libro nella tabella **Books**, impostando automaticamente la sua chiave esterna al valore corretto.

### 6.6.2 Modificare un'associazione

Si modifica un'associazione quando viene cambiata l'*entità* padre di un'*entità* figlia. (Ad esempio, si modifica il genere di un libro.) Anche in questo caso esistono tre strade.

1. Aggiungere l'*entità* figlia alla *collection navigation property* della nuova *entità* padre.
2. Impostare la *reference navigation property* dell'*entità* figlia sulla nuova *entità* padre.
3. Impostare a la proprietà di FK sulla PK della nuova *entità* padre.

### 6.6.3 Rimuovere un'associazione

Rimuovere un'associazione significa eliminare il riferimento all'*entità* padre; è possibile soltanto se si tratta di un'associazione parziale. Anche in questo caso, esistono più strade:

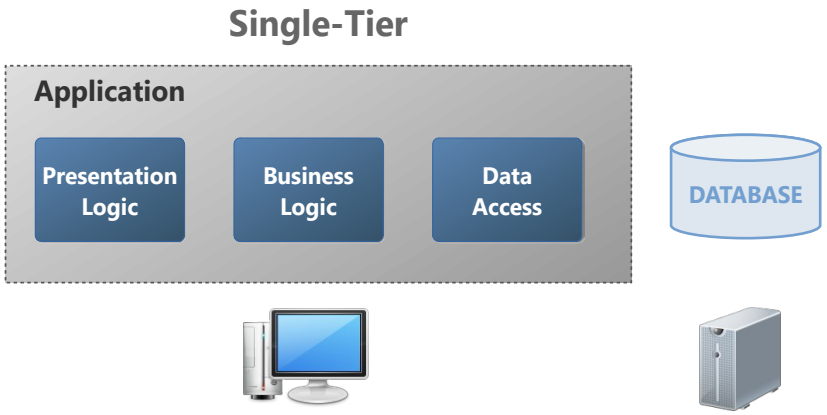
1. Rimuovere l'*entità* figlia dalla *collection navigation property* dell'*entità* padre.
2. Impostare a null la *reference navigation property* dell'*entità* figlia.
3. Impostare a null la FK (se definita) dell'*entità* figlia.

# 7 Uso di EF in scenari “N-Tier”

Negli esempi presentati finora ho dato per scontato che le *entità* fossero tracciate dall'oggetto *context*, cosa che rende semplice gestire la modifica dei dati. Si tratta di una situazione tipica delle applicazioni ***single-tier***, che riepilogo di seguito.

## 7.1 EF e scenari single-tier

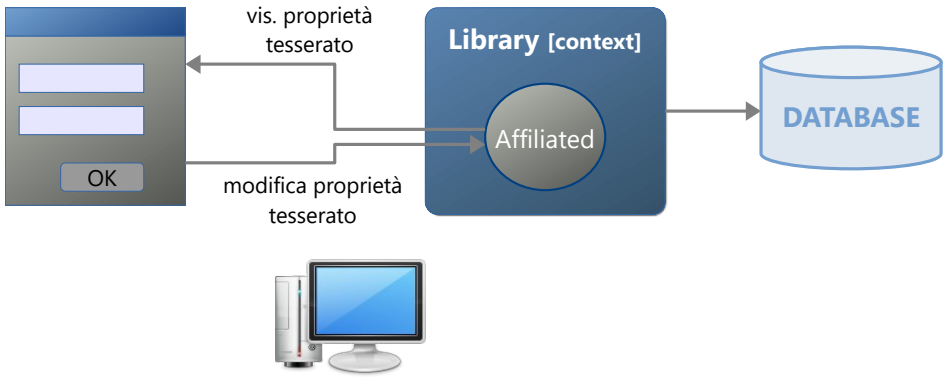
In uno scenario ***single-tier*** l'intero codice applicativo gira all'interno dello stesso processo. La distinzione tra UI (*presentation logic*) e logica di elaborazione e accesso ai dati, (*business logic* + d*ata access*) è puramente logica.




In questo caso è possibile usare lo stesso *context* sia per leggere i dati dal database che per persistere le modifiche.

### 7.1.1 Modifica di un'entità in uno scenario single-tier

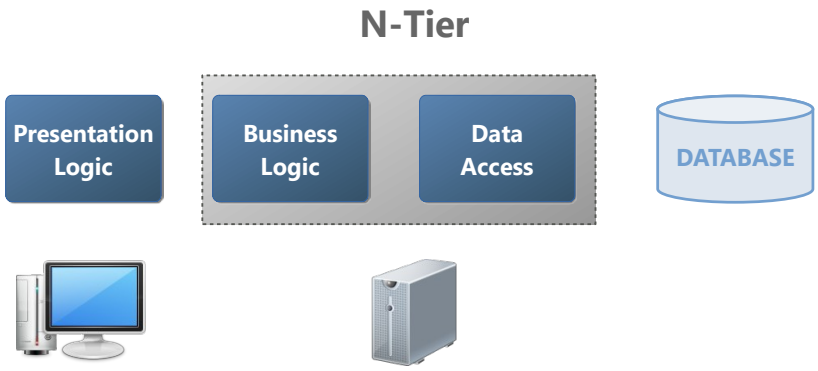
Considera la modifica dei dati di un tesserato. Il tesserato viene caricato dal database e i suoi dati vengono visualizzati in un *form*, consentendo all'utente di modificarli. Il tesserato, così modificato, viene persistito sul database.




Se si utilizza lo stesso *contex*, la persistenza delle modifiche non pone alcun problema; infatti, il *context* “sa” quali proprietà sono state modificate ed è in grado di generare la corretta istruzione SQL.

## 7.2 EF e scenari n-tier

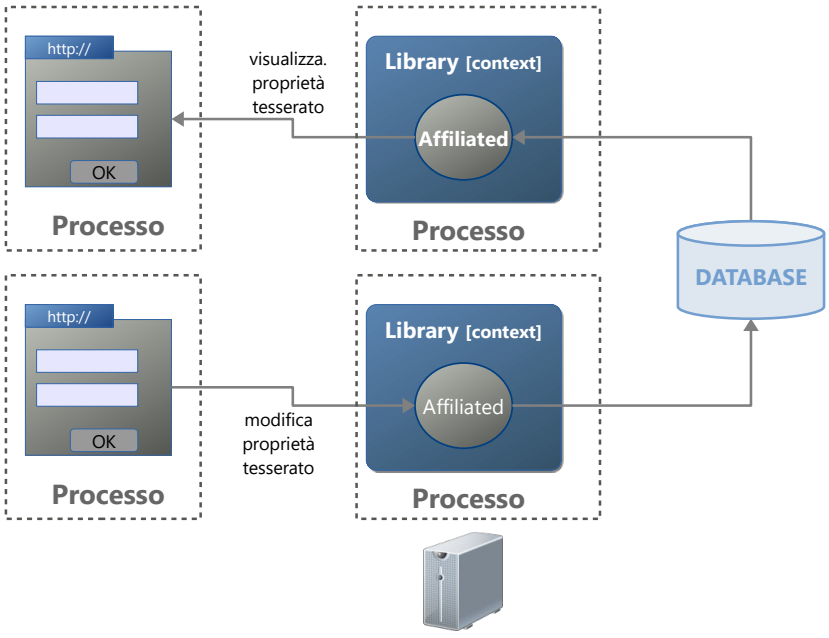
In uno scenario ***n-tier*** l'applicazione è divisa in "strati" che girano su processi diversi, spesso su macchine separate. L'esempio tipico è rappresentato dalle applicazioni web. Il server che ospita l'applicazione elabora le richieste del client, recupera i dati dal database e li spedisce nuovamente al client, il quale li presenta all'utente. Evidentemente: il processo client (il browser) e il processo server girano su macchine distinte.




### 7.2.1 Modifica di un'entità in uno scenario n-tier

Considera l'esempio proposto in (7.1.1), ma stavolta in un contesto client-server. Il client richiede l'operazione di modifica. Il server *avvia un processo* che elabora la richiesta, caricando il tesserato e inviandolo al client. Dopo le modifiche dell'utente, il client invia i dati al server, il quale *avvia un nuovo processo* che persiste i dati sul database.

In questo caso, il *context* usato per caricare il tesserato dal database *non è lo stesso oggetto usato per persistere le modifiche!* Per il secondo *context*, il tesserato rappresenta una nuova *entità* e non un'entità esistente che ha subito delle modifiche; dunque non è in grado di generare la corretta istruzione SQL.




In questo caso il tesserato modificato rappresenta un'***entità*** **disconnessa**, poiché non è stata precedentemente tracciata dall'oggetto *context* che dovrà elaborarla.

## 7.3 Usare EF in scenari “disconnessi”

La gestione di *entità disconnesse* è tipica degli scenari *n-tier*, ma riguarda anche gli scenari *single-tier*. Si parla dunque di:

1. **Scenario connesso**: viene utilizzato lo stesso oggetto *context* per il caricamento delle entità e il salvataggio delle modifiche.
2. **Scenario disconnesso**: vengono usati oggetti *context* distinti.

Di seguito saranno presentati alcuni esempi di *scenari disconnessi*. Per semplicità, la “natura disconnessa” sarà simulata utilizzando un *context* per caricare i dati e un secondo *context* per persistere le modifiche.

## 7.4 Inserimento di una nuova entità

Nei casi più semplici, l'inserimento di un'*entità disconnessa* non richiede una gestione particolare, poiché una nuova *entità* è disconnessa per definizione: basta eseguire il metodo `Add()` e salvare le modifiche.

Il metodo seguente aggiunge un nuovo autore:

```
void AggiungiAutore(Author autore)
{
    var db = new Library();
    db.Authors.Add(autore);
    db.SaveChanges();
}
```

### 7.4.1 Gestire l'inserimento di un “grafo di entità”

Più *entità* associate tra loro rappresentano un ***grafo di entità***. Negli scenari disconnessi questo pone un problema, poiché occorre “istruire” EF sullo stato delle *entità* associate a quella che si vuole inserire.

Considera l’inserimento di un nuovo libro. Durante la procedura di inserimento dati viene selezionato il genere di appartenenza del libro. In fase di inserimento, il server riceve una nuova *entità* `Book` che referenzia (mediante `GenreId`) un'*entità* `Genre` già esistente nel database. Il codice seguente simula l'intero processo:

```
// SERVER: caricamento del genere 'Fantascienza'
var db = new Library();
var genere = db.Genres.Find(1);

// CLIENT: creazione del nuovo libro e associazione al genere 'Fantascienza'
var libro = new Book
{
    Title = "Cronache della Galassia",
    PublicationDate = DateTime.Parse("12/1/1984"),
    Genre = genere,
    AvailableCount = 1
};

// SERVER: inserimento del nuovo libro
AggiungiLibro(libro);
...
```

```
void AggiungiLibro(Book libro)
{
    var db = new Library();
    db.Books.Add(libro);
    db.SaveChanges();
}
```

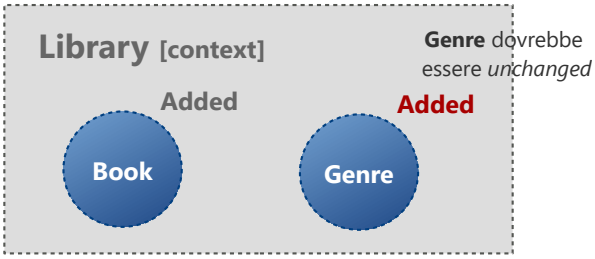
L'esecuzione fallisce, poiché EF tenta di inserire anche il genere, ma questo esiste già.[6]

```
Unhandled exception. Microsoft.EntityFrameworkCore.DbUpdateException: An error occurred while saving the entity changes. See the inner exception for details.
 ---> Microsoft.Data.SqlClient.SqlException (0x80131904): Cannot insert explicit value for identity column in table 'Genres' when IDENTITY_INSERT is set to OFF.
```

Il problema è prodotto dall'istruzione:

```
db.Books.Add(b);
```

che registra sia il libro che il genere come nuove *entità*:




È un comportamento generale: nel modificare lo stato di un'*entità*, EF lo fa anche per tutte le entità associate delle quali non conosce lo stato.

## 7.5 Modificare lo stato delle entità: metodo Entry()

Non esiste una strategia generale, ma, qualsiasi approccio si decida di utilizzare, resta necessario modificare esplicitamente lo stato delle *entità*. Il metodo `Entry()` consente sia di ottenere lo stato di un'entità (vedi 6.1.1), che di modificarlo.

### 7.5.1 Modificare lo stato da Added a Unchanged

Riconsidera l'esempio precedente; perché possa funzionare è necessario informare il *context* che il genere rappresenta un'*entità* esistente e dunque non deve essere inserito nel database.

```
void AggiungiLibro(Book libro)
{
    var db = new Library();
    db.Entry(libro.Genre).State = EntityState.Unchanged;
    db.Books.Add(libro);
```

6 Per chiarezza, l'errore mostrato fa riferimento a un problema diverso. Ma la questione resta la stessa: il codice è destinato a fallire, poiché esiste già un genere con quella chiave primaria.

```
    db.SaveChanges();
}
```

Alternativamente, è possibile usare il metodo `Attach()` per registrare il genere con lo stato *Unchanged*:

```
void AggiungiLibro(Book libro)
{
    var db = new Library();
    db.Books.Add(libro);
    db.Genres.Attach(libro.Genre);
    db.SaveChanges();
}
```

In entrambi i casi, EF si limita a generare una INSERT per aggiungere il nuovo libro, senza eseguire operazioni sulla tabella **Genres**.

### 7.5.2 Distinguere tra entità nuove e già esistenti: verifica della PK

Il codice proposto nell'esempio precedente parte dall'assunzione che il genere sia già esistente nel database; non si tratta di una assunzione sempre valida, poiché il client potrebbe aver effettivamente creato un nuovo libro appartenente a un nuovo genere. Serve un modo per distinguere un'entità nuova da una già esistente nel database.

Un metodo è quello di verificare il valore della chiave primaria. Se vale zero significa che l'*entità* è stata appena creata, altrimenti che è stata caricata dal database:

```
void AggiungiLibroOk3(Book libro)
{
    var db = new Library();
    db.Books.Add(libro);
    if (libro.Genre.GenreId > 0)
        db.Entry(libro.Genre).State = EntityState.Unchanged;
    db.SaveChanges();
}
```

Nota bene: se la chiave primaria del genere è zero, il suo stato viene lasciato su *Added*.

Il seguente codice crea un nuovo libro appartenente a un nuovo genere:

```
// CLIENT: creazione del nuovo libro e genere
var genere = new Genre { Name = "Astrofisica" };
var libro = new Book { Title = "I primi tre minuti", Genre = genere, ... };

// SERVER: inserimento nuovo libro
AggiungiLibro(libro);
```

L'esecuzione produce nell'ordine:

1. L'inserimento del nuovo genere.
2. Il recupero della chiave primaria del genere appena inserito.

3. La valorizzazione della chiave esterna del libro con la chiave primaria del genere.
4. L'inserimento del nuovo libro (e recupero della sua chiave primaria).

### 7.5.3 Caricamento dal database dell'entità associata

La tecnica precedente è semplice, ma non sempre utilizzabile. Ipotizziamo, ad esempio, che dal client venga inviato il libro e il nome del genere. Per sapere se si tratta di un nuovo genere occorre eseguire una query sul database:

```
void AggiungiLibro(Book libro, string nomeGenere)
{
    var db = new Library();
    var genere = db.Genres.Where(g => g.Name == nomeGenere).SingleOrDefault();
    if (genere == null)
        genere = new Genre { Name = nomeGenere };
    libro.Genre = genere;
    db.Books.Add(libro);
    db.SaveChanges();
}
```

Ci sono alcune osservazioni da fare:

1. il metodo `SingleOrDefault()` ritorna il genere cercato, oppure `null` se il genere non esiste.
2. Non viene modificato lo stato del genere. Infatti, o è stato caricato dal database, e dunque si trova già nello stato *Unchanged*, oppure viene creato.
3. Il genere (nuovo o esistente) viene associato al libro. Questo è necessario, poiché dal client proviene un libro senza riferimento al genere.

Segue il codice che simula il processo di inserimento del nuovo libro.

```
// CLIENT: creazione del nuovo libro
var book = new Book { Title = "I primi tre minuti", ...};

// SERVER: inserimento nuovo libro, specificando il nome del genere
AddBook(book, "Astrofisica");
```

## 7.6 Modificare un'entità

La modifica di un'*entità disconnessa* richiede che il suo stato sia impostato su *Modified*, in modo che EF possa generare l'istruzione SQL UPDATE.

Il codice seguente mostra un metodo che riceve un libro, che si suppone già modificato nel client, e lo salva nel database:

```
// SERVER: caricamento del libro
var db = new Library();
var libro = db.Books.Find(1);
```

```
// CLIENT: modifica la data di pubblicazione
libro.PublicationDate = DateTime.Parse("1/1/2011");

// SERVER: aggiorna il libro nel database
AggiornaLibro(libro);
...
void AggiornaLibro(Book libro)
{
    var db = new Library();
    db.Entry(libro).State = EntityState.Modified;
    db.SaveChanges();
}
```

La modifica dello stato dell'entità implica l'aggiornamento di tutte le sue proprietà, come dimostra l'istruzione SQL generata da EF:

```
UPDATE Books
    SET Title = @p0, PublicationDate = @p1, AvailableCount = @p2, GenreId = @p3
WHERE (BookId = @p4)

@p0=N'Fondazione', @p1='2011-01-01 00:00:00', @p2=1, @p3=1, @p4=1
```

### 7.6.1 Modificare un'associazione

Come detto in 6.6.2, questo risultato può essere ottenuto in vari modi, agendo sulle *proprietà di navigazione* o direttamente sulle colonne di chiave esterna. In uno scenario *n-tier* la situazione è complicata dal fatto che il *context* non ha nessuna informazione sulle *entità*.

Di seguito viene esteso l'esempio precedente, considerando la possibilità di cambiare anche il genere del libro. (Per semplificare il codice, si assume che il genere sia già presente nel database).

```
// SERVER: caricamento del libro
var db = new Library();
var libro = db.Books.Find(1);
var genere = db.Genres.Where(g => g.Name == "Fantasy").Single();

// CLIENT: modifica del libro
libro.PublicationDate = DateTime.Parse("1/1/2011");

// SERVER: modifica del libro nel database
AggiornaLibro(libro, genere);
...

void AggiornaLibro(Book libro, Genre genere)
{
    var db = new Library();
    db.Entry(libro).State = EntityState.Modified;
    db.Entry(genere).State = EntityState.Unchanged;
    libro.Genre = genere;
    db.SaveChanges();
}
```

Nota bene: partendo dall'assunzione che il genere sia esistente, occorre modificare il suo stato in *Unchanged*.

## 7.7 Eliminazione di un'entità

In uno scenario "connesso", dove le *entità* sono tracciate, l'eliminazione di un'*entità* si ottiene eseguendo il metodo `Remove()`; EF registra l'*entità* per l'eliminazione, oppure la rimuove dalla memoria se era nello stato *Added*. Con un'*entità disconnessa* ciò non è possibile, poiché EF non può sapere se si tratta di un'*entità* esistente nel database oppure no. La soluzione è quella di registrare l'entità per l'eliminazione usando il metodo `Entry()`.

Il codice seguente elimina un prestito:

```
// SERVER: caricamento del prestito (il primo non ancora restituito)
var db = new Library();
var prestito = db.Loans.Where(p => p.ReturnDate == null).First();

// SERVER: eliminazione del prestito
EliminaPrestito(prestito);
...
void EliminaPrestito(Loan prestito)
{
    var db = new Library();
    db.Entry(prestito).State = EntityState.Deleted;
    db.SaveChanges();
}
```

## 7.8 Scenari disconnessi e *lazy loading*

Esiste una problematica riguardante gli scenari disconnessi e la tecnica del *lazy loading*. (Vedi 4.6) Questa consente di caricare un'*entità associata* soltanto quando si accede ad essa. Perché possa funzionare, però, è necessario che l'entità si connessa a un *context*.

Ad esempio, il seguente codice carica un libro dal database, rilascia il *context* e quindi visualizza gli autori.

```
var db = new Library();
Book libro = db.Books.Find(1);
db.Dispose();                    // il context viene chiuso
VisualizzaAutoriLibro(libro);

...

void VisualizzaAutoriLibro(Book libro)
{
    foreach (var autore in libro.Authors)  // <- qui viene prodotto un errore!
    {
        Console.WriteLine(autore.LastName + ", " + autore.FirstName);
    }
}
```

Il codice evidenziato produce il seguente errore:

```
Unhandled exception. System.InvalidOperationException: An error was generated for warning
'Microsoft.EntityFrameworkCore.Infrastructure.LazyLoadOnDisposedContextWarning': An attempt was
made to lazy-load navigation 'BookProxy.Authors' after the associated DbContext was disposed.
```

L'entità `book` non è più connessa al *context* e dunque il tentativo di caricare gli autori dal database genera un errore. In questo caso, o si attacca il libro a un nuovo *context*, oppure si adotta la tecnica dell'*eager loading* o *dell'explicit loading,* caricando gli autori prima di elaborare il libro. (Vedi 4.4)

## 7.9 Migliorare le performance di caricamento: AsNoTracking()

Esistono molte situazioni nelle quali non è necessario tracciare le *entità* caricate dal database, perché non sono previste modifiche, o perché, come negli scenari *n-tier*, le eventuali modifiche sono gestite da un altro *context*.

Disabilitare il tracciamento delle entità migliora le prestazioni, sia per quanto riguarda la velocità di caricamento, che per la quantità di memoria occupata. A questo scopo si utilizza il metodo `AsNoTracking()` al termine della query.

Il seguente codice carica tutti i libri, ma senza tracciarli. Dopo il caricamento, ognuno di essi si trova nello stato *Detached*:

```
var db = new Library();
var books = db.Books.AsNoTracking();        // tutti i libri sono "Detached"
// elabora i libri
```

## 7.10 Conclusioni sugli scenari *n-tier*

Lo scopo degli esempi presentati è quello di introdurre le problematiche relative all'uso di EF in scenari *n-tier*. Occorre precisare che quelle proposte sono tutte soluzioni ad hoc, poiché il server espone un metodo per ogni operazione richiesta dall'applicazione, la quale definisce esattamente il tipo di modifica effettuata.

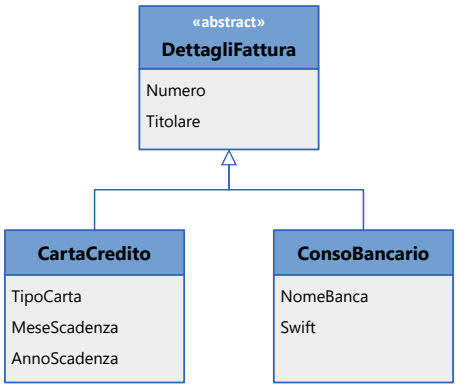
In scenari realistici esistono altri approcci, come ad esempio le ***Self Tracking Entities***, *entità* contenenti la logica necessaria per tracciare le modifiche alle quali sono sottoposte. In questo modo lo stato dell'entità può essere impostato direttamente dal client. Lato server è sufficiente esporre un unico metodo di aggiornamento, utilizzabile per persistere qualsiasi tipo di modifica.

# 8 Gestire le associazioni di generalizzazione

Nella progettazione di un database, una problematica che occorre spesso affrontare è quella di implementare le *associazioni di generalizzazione*. Esistono tre strategie (l'attuale versione di EF consente di adottare le prime due soltanto[7]):

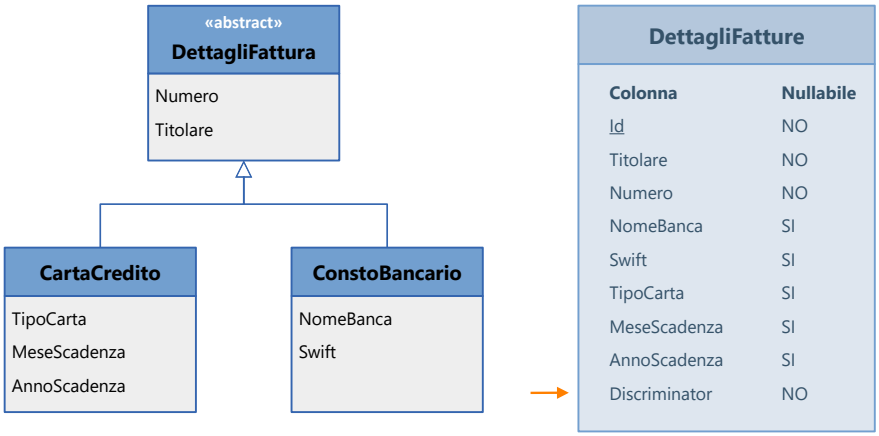
1. **Table per Hierarchy** (TPH)
2. **Table per Type** (TPT).
3. **Table per Concrete class** (TPC).

Qui le analizzerò prendendo ad esempio un sistema di pagamento di una fattura, alla base del quale c'è un'entità astratta, `DettagliFattura`, che definisce i dati comuni ai modi di pagamento: il `Numero` progressivo della fattura e il `Titolare` dell'acquisto. Da `DettagliFattura` derivano i sistemi di pagamento effettivi: `CartaCredito` e `ContoBancario`.




## 8.1 Table Per Hierarchy

La strategia TPH è quella adottata di default da EF. Con questa strategia l'intera gerarchia viene mappata in una sola tabella del database.




In pratica si *denormalizza* il sistema di pagamento. La tabella definisce una colonna, **Discriminator**, che funge da *discriminatore*, poiché consente di identificare ogni record come appartenente a un determinato sistema di pagamento.

7 La versione 7 di EF implementa anche la terza.

Di default, la colonna è stringa e memorizza il nome dell'*entity class* corrispondente al tipo di pagamento ("**CartaCredito**" o "**ContoBancario**", nell'esempio).

La strategia TPH ha un preciso vincolo, che è anche il suo difetto principale: tutte le colonne corrispondenti alle proprietà delle classi derivate devono *devono essere non richieste!*

### 8.1.1 Entity model e classe context nella strategia TPH

Segue l'*entity model* corrispondente al diagramma della pagina precedente:

```
public class FatturazioneContext: DbContext
{
    ...
    public DbSet<DettagliFattura> DettagliFatture { get; set; }
    public DbSet<CountoBancario> ContiBancari { get; set; }
    public DbSet<CartaCredito> CarteCredito { get; set; }
}

...
public abstract class DettagliFattura
{
    public int Id { get; set; }
    public string Titolare { get; set; }
    public string Numero { get; set; }
}

public class CountoBancario : DettagliFattura
{
    public string NomeBanca { get; set; }
    public string Swift { get; set; }
}

public class CartaCredito : DettagliFattura
{
    public int? TipoCarta { get; set; }
    public string MeseScadenza { get; set; }
    public string AnnoScadenza { get; set; }
}
```

Ci sono alcune osservazioni da fare:

1. La classe *context* definisce un *dbset* per ogni *entity class*. È necessario per "informare" EF che anche le classi `CartaCredito` e `ContoBancario` fanno parte dell'*entity model*.
2. Solo la classe `DettagliFattura` definisce la chiave primaria; le classi derivate la ereditano.
3. La proprietà `TipoCarta` è di tipo `int?` (*nullable int)*. Dato il vincolo della strategia TPH, le proprietà *value type* delle entità derivate devono essere *nullabili*.
4. Nel modello non esiste un campo discriminatore, poiché è gestito in modo trasparente da EF.

### 8.1.2 Mappare le entità derivate senza definire dei dbset

Spesso le gerarchie di classi vengono usate in modo polimorfico (8.3); in questo caso, l'unico *dbset* necessario è quello relativo alla classe astratta che si trova alla base della gerarchia. Ciò detto, è possibile evitare la dichiarazione dei *dbset* relativi alle classi derivate mappandole nel metodo `OnModelCreating()`:

```
public class FatturazioneContext: DbContext
{
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<ContoBancario>();
        modelBuilder.Entity<CartaCredito>();
    }

    public DbSet<DettagliFattura> DettagliFatture { get; set; }
}
```

Il metodo `Entity()` fornisce una modalità avanzata di configurazione, utile quando si vuole personalizzare la colonna utilizzata come *discriminatore*, nel nome, nel tipo e nei valori assegnati per ogni entità derivata:

```
public class FatturazioneContext: DbContext
{
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<DettagliFattura>()
            .HasDiscriminator<string>("TipoPagamento")
            .HasValue<ContoBancario>("Conto bancario")
            .HasValue<CartaCredito>("Carta credito");
    }

    public DbSet<DettagliFattura> DettagliFatture { get; set; }
}
```

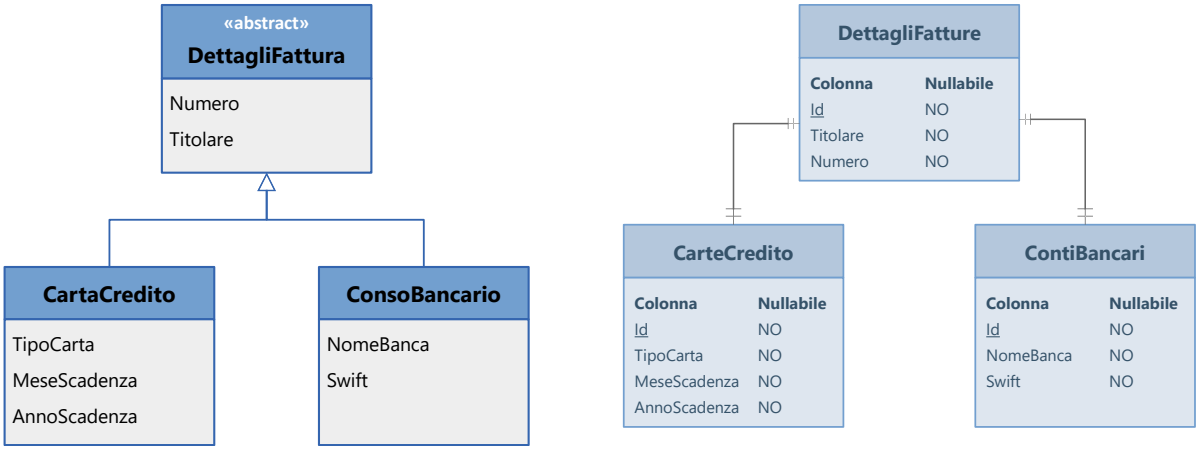
Il codice evidenziato ha una duplice funzione: configura il nome e i possibili valori della colonna *discriminatore* e introduce nell'*entity model* le classi `ContoBancario` e `CartaCredito`.

(Nota bene: le stringhe assegnate alla colonna discriminatore non devono necessariamente coincidere con il nome delle *entity class* corrispondenti.)

## 8.2 Table Per Type

Nella strategia TPT a ogni *entity class* corrisponde una propria tabella, che definisce soltanto le colonne corrispondenti alle proprietà non ereditate. Si tratta di una struttura normalizzata: tra le tabelle derivate e la tabella base esiste un'associazione 1↔1, nella quale la chiave primaria della tabella padre diventa chiave primaria e chiave esterna nelle tabelle figlie.

Poiché ogni *entity class* è mappata con una tabella, non esiste l'obbligo di definire opzionali quei campi che per loro natura non lo sarebbero, come ad esempio `TipoCarta`.




Quando viene creato un nuovo sistema di pagamento, ad esempio una carta di credito, viene inserito un record sia in **DettagliFatture** che in **CarteCredito**. Le colonne **Titolare** e **Numero** vengono inserite nella prima tabella, mentre le colonne **TipoCarta**, **MeseScadenza** e **AnnoScadenza** vengono inserite nella seconda.

### 8.2.1 Entity model e classe context nella strategia TPT

Valgono le regole stabilite in precedenza, con in più la necessità di definire esplicitamente i nomi delle tabelle corrispondenti alle classi derivate. (In caso contrario viene adottata la strategia TPH.)

Ciò può essere fatto mediate l'attributo `[Table]` (5.1), oppure all'interno del metodo `OnModelCreating()`, utilizzando il metodo `Entity()`.

Di seguito propongo la seconda modalità:

```
public class FatturazioneContext: DbContext
{
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<ContoBancario>().ToTable("ContiBancari");
        modelBuilder.Entity<CartaCredito>().ToTable("CarteCredito");
    }

    public DbSet<DettagliFattura> DettagliFatture { get; set; }
}
```

## 8.3 Gestione "polimorfica" delle entità

Per evitare un proliferare di *dbset*, nella classe *context* è possibile definire soltanto il *dbset* corrispondente all'*entity class* alla base della gerarchia. In questo caso, però, è necessario scrivere del codice che stabilisca, per ogni record elaborato, la possibilità che sia un `ContoBancario` o un `CartaCredito`.

Il seguente esempio visualizza l'elenco dei sistemi di pagamento. Per ogni sistema viene visualizzato il proprietario e il tipo di pagamento:

```
var db = new FatturazioneContext();
foreach (var df in db.DettagliFatture)
{
    string tipoPagamento = (df is ContoBancario) ? "Conto bancario" : "Carta di credito";
    Console.WriteLine($"{df.Titolare,-25} - {tipoPagamento}");
}
```

Nota bene: per ogni elemento viene verificato il tipo effettivo mediante l'operatore `is`.

## 8.4 Query non polimorfiche

La gestione polimorfica è problematica quando si desidera elaborare soltanto le *entità* di un determinato tipo, poiché nell'*entity model* non esiste un campo che funga da discriminatore del tipo di *entità*. La soluzione consiste nell'eseguire una query mediante il metodo `OfType<>()`, il quale consente di specificare il tipo di *entità* da caricare.

Il codice seguente visualizza soltanto i sistemi di pagamento che usano un conto bancario:

```
var db = new FatturazioneContext();
foreach (var cb in db.DettagliFatture.OfType<ContoBancario>())
{
    Console.WriteLine(cb.Titolare);
}
```

Naturalmente, il metodo `OfType()` è inutile se la classe *context* definisce un *dbset* corrispondente al sistema di pagamento che si vuole elaborare.

```
var db = new FatturazioneContext();
foreach (var cb in db.ContiBancari)
{
    Console.WriteLine(cb.Titolare);
}
```

http://weblogs.asp.net/manavi/archive/2010/12/24/inheritance-mapping-strategies-with-entity-framework-code-first-ctp5-part-1-table-per-hierarchy-tph.aspx

https://devblogs.microsoft.com/dotnet/announcing-ef7-preview5/

# Appendice I: installare Entity Framework

EF deve essere istallato per ogni progetto che ne fa uso. A questo scopo si usa il **Nuget Package Manager**, oppure la **Package Manager Console**.

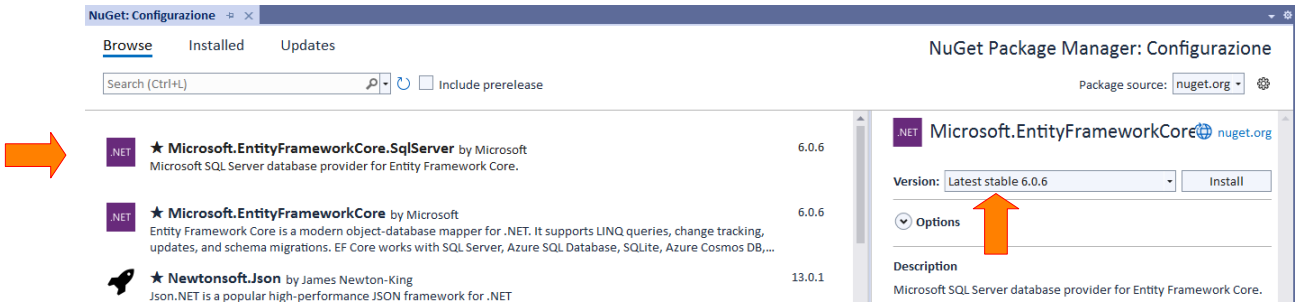
EF consiste di diversi componenti, dei quali soltanto due sono strettamente necessari:

- Microsoft.EntityFrameworkCore
- Microsoft.EntityFramework<*nome provider*>

Basta istallare il secondo, perché venga istallato anche il primo.

Supponi quindi di dover usare EF con un database SQL Server.

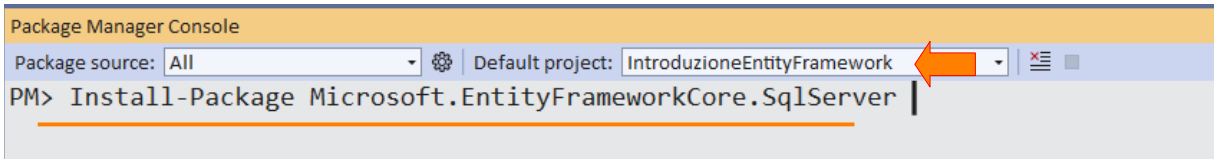
Esegui il **Nuget Package Manager** (**Click destro sul progetto | Manage NuGet Packages…**) e, nella scheda **Browse**, digita Entity Framework nella casella di ricerca e seleziona il provider SQL Server nell'elenco dei pacchetti proposti:




Verifica che la versione del pacchetto (freccia a destra) non sia superiore alla versione del .NET istallato nel computer (e a scuola). Se così è, clicca sull'elenco a discesa e seleziona una versione precedente.

## 8.5 Uso della Package Manager Console

Esegui la console mediante **Tools | NuGet Package Manager | Package Manager Console** e inserisci il comando per istallare il provider SQL Server:




Se la tua *solution* ha più progetti, verifica che sia selezionato il progetto giusto in *default project.*

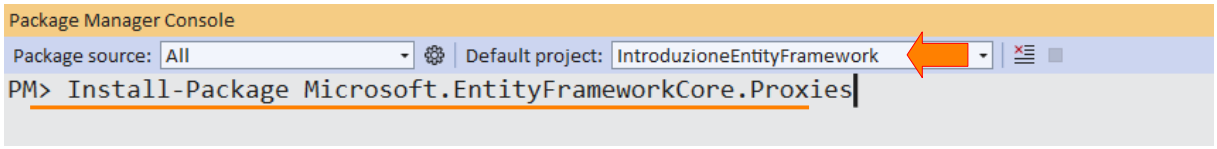
Per istallare una specifica versione, usa l'opzione **-version** e specifica il numero di versione:

```
Install-Package Microsoft.EntityFrameworkCore.SqlServer -version 6.0.6
```

# Appendice II: istallare e configurare il *lazy loading*

Per usare il *lazy loading* occorre istallare il pacchetto: **Microsoft.EntityFrameworkCore.Proxies**. (4.6)

Esegui il comando di menù **Tools | NuGet Package Manager | Package Manager Console** e, nella finestra, inserisci:




Se la tua *solution* ha più progetti, verifica che sia selezionato il progetto giusto in *default project*.

Per istallare una specifica versione, usa l'opzione **-version** e specifica il numero di versione:

```
Install-Package Microsoft.EntityFrameworkCore.Proxies -version 6.0.6
```

## 8.6 Configurare l'uso del lazy loading

La funzione *lazy loading* deve essere abilitata nel metodo `OnConfiguring()` della classe *context*:

```
class Library: DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb...");
        optionsBuilder.UseLazyLoadingProxies();
    }
    ...
}
```

Nota bene: non abilitare il *lazy loading* non pregiudica l'uso di EF; semplicemente, l'accesso alle proprietà di navigazione non produce il caricamento automatico delle *entità* correlate.

## 8.7 Definire virtuali tutte le proprietà di navigazione

Se abiliti il *lazy loading* devi dichiarare virtuali tutte le proprietà di navigazione di tutte le *entity class*. Ciò va fatto indipendentemente dal fatto che la funzione si effettivamente usata oppure no.

# Appendice III: stringa di connessione esterna

Negli esempi mostrati, la *stringa di connessione* è impostata all'interno della classe *context*, nel metodo `OnConfiguring()` (2.3.3). Si tratta di un approccio non realistico; in genere le risorse esterne di un programma (e una *stringa di connessione* lo è) non sono codificate nelle classi che le usano, ma vengono fornite dall'esterno, ad esempio caricate da un file di configurazione.

EF implementa un meccanismo di configurazione sofisticato, che consente di stabilire il *provider* (2.1.1) da utilizzare e le informazioni necessarie al *provider* per referenziare il database. Si tratta però di un meccanismo complicato da mettere in pratica in un'applicazione console. Qui proporrò un approccio più semplice, con l'obiettivo di:

- Implementare una classe *context* che accetti nel costruttore la *stringa di connessione*.
- Caricare la *stringa di connessione* da un file di configurazione utilizzando un pattern consolidato che vede l'uso di un componente ad hoc.

## 8.1 Accettare la stringa di connessione nel costruttore

Questa nuova versione della classe `Library` accetta la *stringa di connessione* e la salva su un campo privato per usarla successivamente nel metodo `OnConfiguring()`:

```
class Library : DbContext
{
    private readonly string cnStr;

    public Library(string cnStr)
    {
        this.cnStr = cnStr;
    }
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(cnStr);
    }

    //... definisce i dbset
}
```

L'uso della classe richiede che venga fornita la stringa di connessione:

```
var db = new Library("Server=(localdb)\\mssqllocaldb; database=Library;...");
```

## 8.2 Memorizzare la stringa in un file di configurazione

Naturalmente potremmo usare un semplice file di testo contenente la stringa di connessione, ma intendo mostrare il pattern standard, che vede l'uso di componenti di configurazione ad hoc e la memorizzazione delle impostazioni di configurazione in un file JSON chiamato **AppSettings.json**.

Innanzitutto è necessario istallare il pacchetto **Microsoft.Extensions.Configuration.Json**:

```
PM> Install-Package Microsoft.Extensions.Configuration.Json
```

Dopodiché si crea un file JSON di nome **AppSettings.json** (in realtà il nome può essere qualsiasi) con il seguente contenuto:

```
{
  "ConnectionStrings": {
    "cnStr": "Server=(localdb)\\mssqllocaldb; database=Library;..."
  }
}
```

Nota bene: la voce `ConnectionString` deve chiamarsi esattamente così, mentre la voce `cnStr` è soltanto una chiave da associare al valore corrispondente, la *stringa di connessione*, appunto.

Infine: il file **AppSettings.json** deve essere configurato per essere copiato nella *cartella di output*.

### 8.2.1 Creazione e uso della configurazione

Invece di accedere direttamente al file, utilizziamo il meccanismo di configurazione implementato dalla classe `ConfigurationBuilder()`:

```
using Microsoft.Extensions.Configuration;   // necessario per usare ConfigurationBuilder
...
var config = new ConfigurationBuilder()     // crea e restituisce l'oggetto di
    .AddJsonFile("AppSettings.json")        // configurazione
    .Build();

var cnStr = config.GetConnectionString("cnStr");
// -> "Server=(localdb)\\mssqllocaldb; database=Library;..."
```

Il primo blocco di codice evidenziato deve essere eseguito una volta per tutte all'avvio del programma, poiché crea l'*oggetto di configurazione*, attraverso il quale si può accedere alle impostazioni memorizzate in **AppSettings.json** specificando la loro chiave.

Esistono svariati modi per accedere alle impostazioni; il metodo `GetConnectionStrings()` è specializzato nell'accesso alle chiave o alle chiavi memorizzate nella sezione **ConnectionStrings**.

# Appendice IV: Proprietà *nullabili* e NRT

I tipi del linguaggio C# possono essere suddivisi in due categorie:

- I *reference type*: sono tipi *nullabili*, perché le variabili possono contenere `null`, dunque non memorizzare alcun valore.
- I *value type*: sono tipi *non-nullabili*, perché le variabili contengono sempre un valore, cioè non possono contenere `null`.

Esempio principe della prima categoria è il tipo `string`; esempio principe della seconda è il tipo `int`. Quanto detto ha delle implicazioni, che riassumo nel seguente codice:

```
string nome = null;   // OK
int altezza = null;   // Errore
...
class Persona
{
    public string Nominativo;  // -> null (valore predefinito)
    public int Età;            // -> 0    (valore predefinito)
}
```

Nel 2006 è stato introdotta una nuova categoria di tipi che unisce le caratteristiche dei *reference type* e dei *value type*: i ***nullable value type***. Questi si ottengono a partire dai *value type* con l'aggiunta del simbolo `?`:

```
string nome = null;   // OK
int altezza = null;   // Errore
int? età = null;      // OK
...
class Persona
{
    public string Nominativo;  // -> null (valore predefinito)
    public int Età;            // -> 0    (valore predefinito)
    public int? altezza;       // -> null (valore predefinito)
}
```

Nel 2019 sono stati introdotti i ***N**ullable **R**eference **T**ype*. Non si tratta di una nuova categoria di tipi, poiché i *reference type* sono già *nullabili*; è una funzione del linguaggio che consente di creare una distinzione tra *reference type nullabili* e *reference type non-nullabili*.

L'idea è quella di poter dichiarare variabili di per sé *nullabili* con la garanzia che non contengano `null` (dunque con la garanzia che contengano un valore). Ci pensa il linguaggio, analizzando il programma (eseguendo una cosiddetta *null reference analysis*), a segnalare il codice che non soddisfa il vincolo di *non-nullabilità*.

Per implementare questa funzione, invece di aggiungere un nuovo simbolo ai tipi *nullabili*, è stato deciso di allineare la sintassi alla scelta fatta nel 2005. In sintesi:

- L'uso del simbolo `?` indica che una variabile è *nullabile*.
- L'assenza del simbolo `?` indica che la variabile, pur appartenente a un tipo *nullabile*, si intende *non-nullabile*.

Il codice seguente riassume la situazione:

```
string nome = null;       // Avvertimento: vincolo non-nullabilità violato!
string? telefono = null; // OK
int altezza = null;       // Errore
int? età = null;          // OK
...
class Persona
{
    public string Nominativo;  // -> null (Avvertimento: vincolo non-nullabilità violato!)
    public string? Email;      // -> null (OK)
    public int Età;            // -> 0    (OK)
    public int? altezza;       // -> null (OK)
}
```

Nota bene:

- La *non-nullabilità* (assenza del simbolo `?`) viene violata non soltanto assegnando `null`, ma anche dichiarando una variabile *non-nullabile* senza assegnarle un valore.
- La violazione del vincolo non rappresenta un errore di programmazione: il compilatore emette un *warning* (avvertimento).

## 8.1 Entity Framework e *Nullable Reference Types*

La gestione della *nullabilità* introdotta con i NRT ha un impatto anche su EF, il quale mappa le proprietà *non-nullabili* con colonne NOT NULL. È come se alle proprietà *reference type*, ma *non-nullabili* (senza `?`), fosse applicato l'attributo `[Required]` (5.10).

Questo comportamento influenza la generazione del database a partire dal modello: Appendice V: generare il database dal modello e il caricamento di record contenenti valori NULL in colonne corrispondenti a proprietà *non-nullabili* (5.10).

Considera questo scenario e la tabella `AuthorsNote`, alla quale corrisponde l'*entity class* `AuthorNote`:

```
[Table("AuthorsNote")]
public class AuthorNote
{
    [Key]
    public int AuthorId { get; set; }      // -> AuthorId  INT IDENTITY (1, 1) NOT NULL
    public string Biography { get; set; }  // -> Biography NVARCHAR(MAX) NULL
    public Author Author { get; set; }
}
```

Basandoci su questa configurazione, il comportamento di EF risulta diverso in base al fatto che la funzione NRT sia abilitata oppure no.

### 8.1.1 Funzione NRT disabilitata

La proprietà `Biography` viene considerata *nullabile* e dunque coerente con la configurazione della colonna omonima. I due modelli sono allineati e non esiste alcun problema nel caricamento dei dati.

### 8.1.2 Funzione NRT abilitata

La proprietà `Biography` viene considerata *non-nullabile*, dunque non coerente con la colonna omonima. Se un record della tabella **AuthorsNote** contiene NULL nella colonna suddetta, il caricamento dei dati produce un'eccezione, poiché EF "si rifiuta" di assegnare `null` alla proprietà `Biography`. Si ottiene in pratica lo stesso comportamento che si otterrebbe se la proprietà fosse decorata con l'attributo `[Required]`.

## 8.2 Conclusioni

Con l'introduzione della funzione NRT si corre il rischio di ottenere un funzionamento diverso del programma in base al fatto che la funzione si attivata oppure no. In uno scenario come il precedente, l'approccio corretto è utilizzare una proprietà *nullabile*:

```
[Table("AuthorsNote")]
public class AuthorNote
{
    [Key]
    public int AuthorId { get; set; }       // -> AuthorId  INT IDENTITY (1, 1) NOT NULL
    public string? Biography { get; set; }  // -> Biography NVARCHAR(MAX) NULL
    public Author Author { get; set; }
}
```

Ora l'*entity model* e lo schema del database sono allineati e non esiste alcun problema di caricamento dei dati.

(Nota finale: se la funzione NRT è disattivata, l'uso di `?` sui *reference type* produce un avvertimento che ci informa sul fatto che il simbolo viene semplicemente ignorato.)

# Appendice V: generare il database dal modello

Il tutorial si basa sull'assunto che il database sia preesistente all'applicazione, ma EF fornisce anche un modello di sviluppo opposto, nel quale è il database ad essere generato automaticamente a partire dall'*entity model*. Ciò consente al programmatore di sviluppare applicazioni senza dover mai utilizzare il linguaggio SQL o uno strumento di progettazione di database.

Questo modello di sviluppo esiste in due forme, la prima delle quali richiede l'istallazione del pacchetto **MicrosoftEntityFrameworkCore.Tools**, è piuttosto sofisticata ed è dedicata a scenari realistici. La seconda forma è semplice e immediata, ed è utile per la realizzazione di un prototipo dell'applicazione, o semplicemente per analizzare le strategie di mapping adottate da EF.

Qui mostro in azione la seconda possibilità.

## 8.3 Creazione automatica del database

EF fornisce la funzione di creazione automatica del database in modo conforme all'*entity model*. La classe `DbContext` fornisce una proprietà, `Database`, che definisce i metodi `EnsureDeleted()` ed `EnsureCreated()`. Il primo garantisce l'eliminazione del database (se il database non esiste, il metodo non produce errori). Il secondo garantisce la creazione del database (se il database esiste già, il metodo non compie alcuna azione e non produce errori.)

L'uso di questi metodi nel costruttore della classe *context* fa sì che alla creazione di un oggetto *context* il database venga creato da zero. Di solito a questo approccio si accompagna il codice necessario per inserire automaticamente dei dati nel database.

### 8.3.1 Sviluppo incrementale dell'entity model

Supponi di realizzare un'applicazione che gestisce i dati sui premi Nobel. L'obiettivo è gestire le categorie di premiazione e le persone che hanno ricevuto il premio. È possibile partire con un modello minimale, che rappresenta i soli premiati:

```
class Nobel : DbContext
{
    public Nobel()
    {
        Database.EnsureDeleted();
        Database.EnsureCreated();
    }

    public DbSet<Premiato> Premiati { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer("...");
    }
}
...
public class Premiato
{
```

```
    public int PremiatoId { get; set; }
    public string Nome { get; set; }
    public string Cognome { get; set; }
}
```

L'esecuzione dell'applicazione produce la creazione di un database contenente la tabella **Premiati**, che definisce una colonna di chiave primaria *identity*:

| Premiati | | |
|---|---|---|
| PremiatoId | INT IDENTITY | NOT NULL |
| Nome | NVARCHAR(MAX) | NOT NULL |
| Cognome | NVARCHAR(MAX) | NOT NULL |

Successivamente si può aggiungere la classe `Premio` e il *dbset* `Premi`. Alla classe `Premiato` si aggiunge una *reference property* alla classe `Premio`, allo scopo di implementare l'associazione tra le due *entity class*.

All'avvio del programma viene creato un nuovo database contenente le tabelle **Premi** e **Premiati**, le quali hanno un'associazione 1↔N. La tabella **Premiati** definisce la colonna di chiave esterna **PremioId** (con relativo vincolo di chiave esterna):

```
public class Premiato
{
    public int PremiatoId { get; set; }
    public string Nome { get; set; }
    public string Cognome { get; set; }
    public Premio Premio { get; set; }
}

public class Premio
{
    public int PremioId { get; set; }
    public string Nome { get; set; }
}
```

| Premiati | | |
|---|---|---|
| PremiatoId | INT IDENTITY | NOT NULL |
| Nome | NVARCHAR(MAX) | NOT NULL |
| Cognome | NVARCHAR(MAX) | NOT NULL |

| Premi | | |
|---|---|---|
| PremioId | INT IDENTITY | NOT NULL |
| Nome | NVARCHAR(MAX) | NOT NULL |

In conclusione, con questo modello di sviluppo ci si può concentrare sulla progettazione del modello e sulle funzioni dell'applicazione, lasciando che sia EF ad occuparsi di generare un database conforme al modello stabilito.

## 8.4 Generazione delle colonne obbligatorie / non richieste

Nel generare il database, EF deve stabilire i vincoli NULL / NOT NULL da applicare alle colonne corrispondenti alle proprietà dell'*entity model*. (Vedi 5.8).

Di default, EF utilizza le regole del linguaggio C#:

- Per una proprietà *nullabile* viene generata una colonna NULL.
- Per una proprietà *non-nullabile* viene generata una colonna NOT NULL.

Sulla base di questo occorre progettare attentamente l'*entity model*. Ad esempio, supponi di aggiungere le date di nascita e di morte dei premiati. Ovviamente, la prima è richiesta, mentre la seconda è opzionale:

```
public class Premiato
{
    ...
    public DateTime DataNascita { get; set; } // -> DataNascita DATETIME2 (7) NOT NULL
    public DateTime? DataMorte { get; set; }  // -> DataNascita DATETIME2 (7) NULL
}
```

### 8.4.1 Generazione colonne richieste corrispondenti alle proprietà nullabili

Prima dell'introduzione della funzione ***N**ullable **R**eference Types* (Appendice IV: Proprietà nullabili e NRT), la regola era semplice: a una proprietà *reference type* (*nullabile*) viene fatta corrispondere una colonna NULL:

```
public class Premiato
{
    public int PremiatoId { get; set; }  // -> PremiatoId  INT IDENTITY (1, 1) NOT NULL
    public string Nome { get; set; }     // -> Nome        NVARCHAR (MAX) NULL
    public string Cognome { get; set; }  // -> Cognome     NVARCHAR (MAX) NULL
    ...
}
```

In questo scenario, se si desidera che una colonna sia richiesta, è necessario usare l'attributo `[Required]`:

```
public class Premiato
{
    public int PremiatoId { get; set; }  // -> PremiatoId  INT IDENTITY (1, 1) NOT NULL

    [Required]
    public string Nome { get; set; }     // -> Nome        NVARCHAR (MAX) NOT NULL

    [Required]
    public string Cognome { get; set; }  // -> Cognome     NVARCHAR (MAX) NOT NULL
    ...
}
```

Con l'introduzione della funzione NRT, EF considera le proprietà *non-nullabili* (senza simbolo `?`) corrispondenti a colonne NOT NULL.

Dunque, considerando l'esempio precedente, anche in assenza dell'attributo `[Required]`, alle proprietà `Nome` e `Cognome` vengono fatte corrispondere delle colonne richieste:

```
public class Premiato
{
    public int PremiatoId { get; set; }  // -> PremiatoId  INT IDENTITY   NOT NULL
    public string Nome { get; set; }     // -> Nome        NVARCHAR (MAX) NOT NULL
    public string Cognome { get; set; }  // -> Cognome     NVARCHAR (MAX) NOT NULL
    ...
}
```

```
public class Premio
{
    public int PremioId { get; set; }    // -> PremioId  INT IDENTITY   NOT NULL
    public string Nome { get; set; }     // -> Nome      NVARCHAR (MAX) NOT NULL
}
```

Se si desidera che a una proprietà *reference type* corrisponda effettivamente una colonna NULL, occorre rendere la proprietà *nullabile* decorando il tipo con `?`:

```
public class Premiato
{
    public int PremiatoId { get; set; }  // -> PremiatoId  INT IDENTITY   NOT NULL
    public string Nome { get; set; }     // -> Nome        NVARCHAR (MAX) NOT NULL
    public string Cognome { get; set; }  // -> Cognome     NVARCHAR (MAX) NOT NULL
    public string? Nazione { get; set; } // -> Nazione     NVARCHAR (MAX) NULL
    ...
}
```

# Appendice VI: *namespaces*

Le funzionalità di Entity Framework sono definite all'interno di vari *namespace*.

### *Microsoft.EntityFrameworkCore*

È il *namespace* che definisce il cuore del framework: classi `DbContext`, `DbSet`, etc.

### *System.ComponentModel.DataAnnotations*

Definisce alcuni attributi utilizzati per configurare il *domain model*, annotando le proprietà delle *entity classes*. Tra questi: `[Key]`, `[MaxLength]` e `[Required]`.

### *System.ComponentModel.DataAnnotations.Schema*

Definisce altri attributi utilizzati per configurare il *domain model*: `[Column]`, `[Table]`, `[NotMapped]`, `[ForeignKey]`.