Entity Framework

Introduzione all'uso di Entity Framework

Progetto di Informatica classe 5^{a}

Anno 2013/2014

Ambiente: .NET 4.5+/C# 5.0+

Entity Framework 5-6

Indice generale

1 Object-Relational Mapping (Mapper)

Il termine ORM viene usato per designare una tecnica di programmazione, **Object-Relational Mapping**, e un tipo di software, **Object-Relational Mapper**. In entrambi i casi 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 il ***domain model*** e il database che:

1. Consente di superare l'incompatibilità tra i modelli *object oriented* e relazionale. (*Impedance mismatch*)
2. Semplifica le operazioni di interrogazione e manipolazione dei dati memorizzati nel database.
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*.

1.1 Funzionalità di un ORM

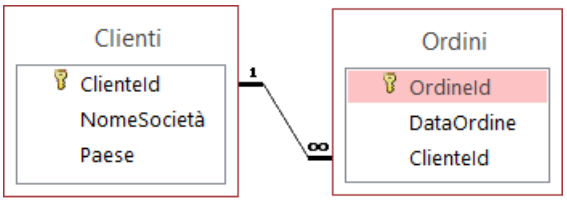
La funzione fondamentale di un ORM è quella di facilitare la gestione del *domain model* attraverso un modello *object oriented*, che viene persistito in un database. Alla base di questa c'è la capacità di "mappare" gli oggetti del modello con le tabelle e le associazioni del database. Ciò consente al codice di applicativo di utilizzare gli oggetti del modello senza dover conoscere dove e in che modo questi vengono persistiti.

> **ORM e storage dei dati**
>
> Qui si da per scontato che l'ORM debba dialogare con un DBMS, ma nella realtà 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"

1.1.1 Esempio di mapping tra modello OO e database

Considera un database per la gestione degli ordini; più precisamente le tabelle Clienti e Ordini:



Tra le varie operazioni ci sarà l'esecuzione di istruzioni SQL per ottenere, ad esempio, l'elenco dei clienti italiani e degli ordini che hanno eseguito. In risposta alla query il DBMS fornisce i dati in forma tabellare:

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

Nel codice applicativo, però, risulta più semplice gestire i dati come un insieme di oggetti. In sostanza, vorremmo poter utilizzare il seguente *domain model*:

```
public class Cliente
{
    public int ClienteId { get; set; }
    public string NomeSocietà { get; set; }
    public string Paese { 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; }
}
```

E poter scrivere il seguente codice:

```
List<Cliente> clientiIta = ... // ottieni i clienti italiani
foreach (var o in clientiIta.Ordini)
{
    Console.WriteLine(o.DataOrdine);
}
```

Ciò è possibile, purché si scriva il codice che interroghi il database, recuperi il result set e costruisca gli oggetti (elenco clienti ed elenco ordini per ogni cliente) del *domain model*. Un ORM è in grado di eseguire automaticamente queste ad altre operazioni.

1.1.2 Caratteristiche generali degli ORM

Gli ORM offrono diverse funzionalità:

1. Generazione automatica del *domain model* a partire dallo schema definito nel database.
2. Generazione e del database a partire dal *domain model*. (Aggiornamento dello schema del database in relazione ai cambiamenti del *domain model*.)
3. Definizione di un linguaggio di interrogazione in grado di operare sul *domain model* e di agire, in modo trasparente, sul database.
4. Gestione della concorrenza, con la definizione di regole per la gestione di conflitti nell'accesso simultaneo ai dati.
5. *Caching,* per ridurre l'accesso al database e dunque migliorare le prestazioni.
6. Implementazione del pattern ***Unit of Work***. Consente di ottimizzare l'accesso al database e di considerare un insieme di modifiche come un'unità, che deve essere completata con successo, oppure essere annullata in caso di errore.

1.2 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.

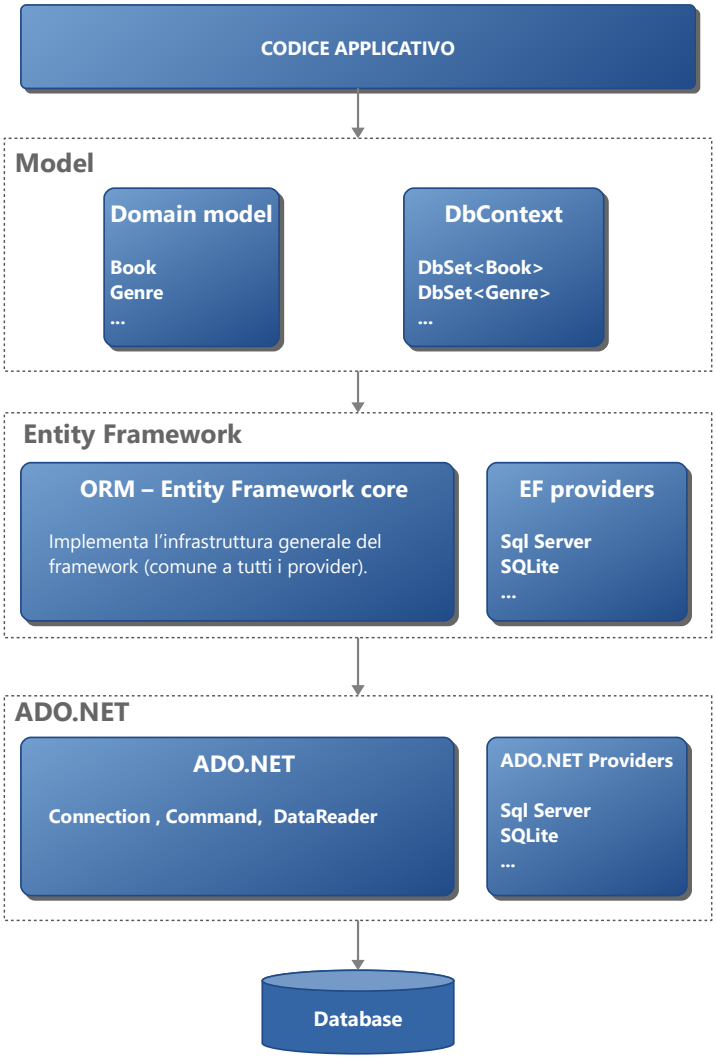
2 Entity Framework

Entity Framework (d'ora in avanti EF) è un ORM sviluppato da Microsoft, parte integrante del .NET Framework, ora in versione open source, scaricabile come pacchetto NuGet.

> **EF 6 e EF Core**
>
> Esistono due versioni distinte di EF. EF 6 è utilizzabile soltanto in ambiente Windows, mentre EF Core (parte della piattaforma .NET Core) è un framework *cross-platform*.
>
> Attualmente, le funzionalità offerte non sono completamente sovrapponibili. In questp tutorial faccio riferimento alla versione EF 6 (anche se la maggior parte delle funzioni mostrate sono utilizzabili anche con EF Core).

2.1 Panoramica su EF

EF nasce come un'estensione di ADO.NET, la cui infrastruttura viene tuttora utilizzata per accedere ai dati. EF consente al codice applicativo di agire sul *domain model*, producendo automaticamente i comandi SQL necessari per agire sul database sottostante.



2.1.1 EF Providers

In sé, EF non è in grado di dialogare con un database specifico o una qualsiasi altra sorgente dati, esattamente come non lo è ADO.NET. A questo scopo esistono gli ***EF Provider***. Questi implementano le funzionalità necessarie per supportare un database specifico. Dunque, EF non è utilizzabile con quei DBMS per i quale non esiste un *EF Provider*, come ad esempio Access.

Gli *EF Provider* si differenziano anche per il livello di supporto che forniscono; questo dipende in parte dalle caratteristiche del DBMS sottostante. Ad esempio, se il DBMS non gestisce le transazioni, nemmeno il provider sarà in grado di farlo. Altre differenze riguardano il supporto all'integrazione con Visual Studio:

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 Domain model (*entity model*)

Il *domain model* sta alla base di qualsiasi applicazione che usa EF e rappresenta l'incarnazione *object oriented* del modello *Entity-Relationship* implementato nel database. In EF il *domain model* viene designato con il termine ***entity model***. Nell'implementazione del modello, EF consente di seguire tre strade distinte, denominate **Model-First**, **Database-First, Code-First**.

2.2.1 Model-First

Questo approccio prevede che venga innanzitutto disegnato l'*entity model*, utilizzando l'*entity framework designer* di Visual Studio. Il risultato è un diagramma UML, a partire dal quale è possibile:

1. Generare un database con uno schema corrispondente al modello.
2. Generare le ***domain classes***, e cioè le classi che costituiscono il modello.

Le informazioni che descrivono il modello sono memorizzate in un file EDMX, suddiviso in tre parti, descritte da altrettanti linguaggi XML. Il modello grafico semplifica la fase progettazione, ma per il programmatore l'obiettivo finale è rappresentato dal codice corrispondente, poiché è questo che sarà utilizzato dall'applicazione e da EF.

2.2.2 Database-First

Si parte dal database e da questo si genera il modello grafico corrispondente, con relativo file EDMX. Da quest'ultimo si generano le classi del modello.

2.2.3 Code-First

Il programmatore ha la responsabilità di scrivere le *domain classes*. Non esiste quindi un file EDMX che descriva il modello. A partire dalle classi è possibile:

1. Generare un nuovo database. Sarà EF a desumere lo schema del database dalla struttura del modello.
2. Mappare un database esistente. Se la struttura delle *domain classes* non corrisponde allo schema del database, EF genera un errore.

3 Code-First

Con Code-First il programmatore ha la responsabilità di definire correttamente le *entity class*, le quali saranno mappate con le tabelle del database.

> **Code-First e generazione del modello**
>
> Esistono tool di "*reverse-engineering*" che, partendo dal database, generano le classi del *domain model* senza passare per il modello EDMX.

3.1 Domain model e entity class

Considera il seguente *domain model*, composto da due *entity classes,* `Cliente` *e* `Ordine`:

```
public class Cliente
{
    public int ClienteId { get; set; }
    public string NomeSocietà { get; set; }
    public string Paese { 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; }
}
```

Il modello ha come caratteristica quella della ***persistence ignorance:*** non c'è alcun riferimento alla modalità di memorizzazione di clienti e ordini (nome database, nomi tabelle, colonne, chiavi, etc).

Le *entity class* vengono definite classi **POCO** (*Plain Old CLR Object*); sono infatti delle normali classi (tipicamente dei *record*) che non devono definire alcuna funzionalità specifica per essere utilizzate con un database.

(Come vedremo più avanti, la *persistence ignorance* completa è difficilmente raggiungibile.)

3.1.1 Entità

Per convenzione, un oggetto del *domain model* viene definito **entità**. Un'entità rappresenta spesso l'analogo di un record nella tabella di un database.

3.2 DbContext

La classe `DbContext` rappresenta il punto d'accesso alle funzionalità di EF. Un `DbContext` (convenzionalmente definito *context*) gestisce le entità del *domain model* e le operazioni eseguite su di esse, e le traduce in operazioni da eseguire sul database sottostante.

`DbContext` deve essere derivata per poter gestire uno specifico *domain model*. Di seguito viene definito viene definita una classe *context* per la gestione del database **Library** :

```
using System.Data.Entity;
...
public class Library: DbContext
{
    public Library(): base("Library")
    {
        Database.SetInitializer<Library>(null);
    }

    public DbSet<Book> Books{ get; set; }
    public DbSet<Genre> Genres { get; set; }
}
```

3.2.1 Impostare la stringa di connessione

La classe *context*, `Library` nell'esempio, deve stabilire la stringa di connessione da utilizzare per accedere al database e passarla alla classe base, `DbContext`. Vi sono due modi:

- Passare alla classe base la stringa di connessione.
- Passare alla classe base la chiave da utilizzare per accedere alla stringa di connessione nel file di configurazione **App.Config**. (come nell'esempio)

Nel secondo caso, il file **App.Config** deve specificare nella sezione **ConnectionStrings** la chiave suddetta e la relativa stringa di connessione:

```
<connectionStrings>
  <add name="Library"
     connectionString="Data Source=MSSQLLocalDB; Initial Catalog=Library;Integrated
                       Security=true; MultipleActiveResultSets=True"
     providerName="System.Data.SqlClient" />
</connectionStrings>
```

3.2.2 DbSet<>

La classe `DbSet` gestisce un insieme di entità; rappresenta il corrispondente di una tabella del database. La maggior parte delle operazioni coinvolge uno o più oggetti `DbSet`.

Il seguente codice visualizza l'elenco dei libri del database, accessibili attraverso la proprietà `Books`, di tipo `DbSet<Book>`:

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

Nota bene: L'iterazione della proprietà `Books` produce il caricamento dei dati dalla tabella `Books` in modo completamente trasparente per il codice applicativo.

3.2.3 Utilizzo dell'oggetto context

L'uso di un *context* può seguire due pattern distinti. Nel primo, l'oggetto viene creato e distrutto per ogni operazione o insieme di operazioni:

Si crea l'oggetto: `Library db = new Library()`

Si eseguono una o più query sulle entità

Si rilascia l'oggetto: `db.Dispose()`

Per garantire l'esecuzione del metodo `Dispose()` anche in caso di eccezioni, è opportuno racchiudere il precedente pattern in un costrutto `using`:

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

Il secondo pattern prevede di utilizzare sempre lo stesso *context*, che sarà creato una sola volta e mai rilasciato, perlomeno nell'ambito di un insieme di operazioni.

Nel resto del tutorial mi limiterò a creare il *context* senza fare riferimento a un pattern preciso.

3.3 Mapping fra *domain model* e schema del database

Il processo di *mapping* consiste nel mettere in corrispondenza il *domain model* con lo schema del database, ed eventualmente sollevare degli errori se una corrispondenza non esiste. Esistono tre strategie, combinabili tra loro: ***convention***, ***annotation*** e ***fluent api configuration***.

3.3.1 Convention

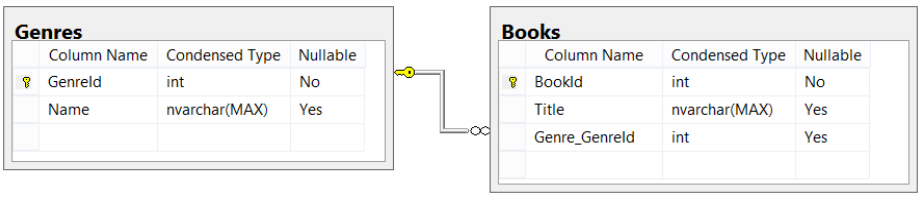
È la strategia predefinita e più semplice. EF usa determinate convenzioni per dedurre lo schema del database a partire dal *domain model*. Queste riguardano i nomi, i tipi di dati e le relazioni tra le entità del modello.

Ad esempio, le classi:

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

public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public Genre Genre { get; set; }       // proprietà di navigazione
}
```

vengono tradotte nelle seguenti tabelle:



Nota bene:

1. Entrambe le tabelle hanno il nome "pluralizzato" della classe corrispondente.
2. I campi `GenreId` e `BookId` vengono utilizzati come chiavi primarie di tipo *identity*.
3. Ai campi stringa corrisponde la colonna omonima di tipo **nvarchar(max)**; colonna che ammette valori nulli.
4. Viene creata una reazione 1↔N tra le tabelle **Genres** e **Books**. La proprietà `Genre` della classe `Books` viene tradotta in una chiave esterna di nome **Genre_GenreId**.

Il solo uso delle convenzioni può essere insufficiente. Innanzitutto, queste sono calibrate per la lingua inglese: un'entità di nome "Libro" viene tradotta in una tabella di nome **Libroes** e non **Libri**. Inoltre, ad esempio, il mapping della proprietà **Genre** produce una FK di nome **Genre_GenreId** e non, come sarebbe più naturale, **GenreId**.

3.3.2 Annotations

Le *annotazioni* sono attributi utilizzati per decorare le classi e i campi del *domain model*. Consentono di specificare ulteriori informazioni e modificare il mapping basato sulle convenzioni. Ad esempio, attraverso l'annotazione `[MaxLength]` è possibile specificare la lunghezza massima di un campo di testo.

```
using System.ComponentModel.DataAnnotations;

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

    [MaxLength(50)]
    public string Title { get; set; }
}
```

L'annotazione sarà usata sia per generare lo schema del database, sia per validare il modello, evitando che possa memorizzare valori che non soddisfino determinati vincoli.

3.3.3 Fluent API configuration

La *Fluent API* designa una modalità di configurazione eseguita a livello di codice. Le opzioni di configurazione viste con le annotazioni sono disponibili anche mediante Fluent API.

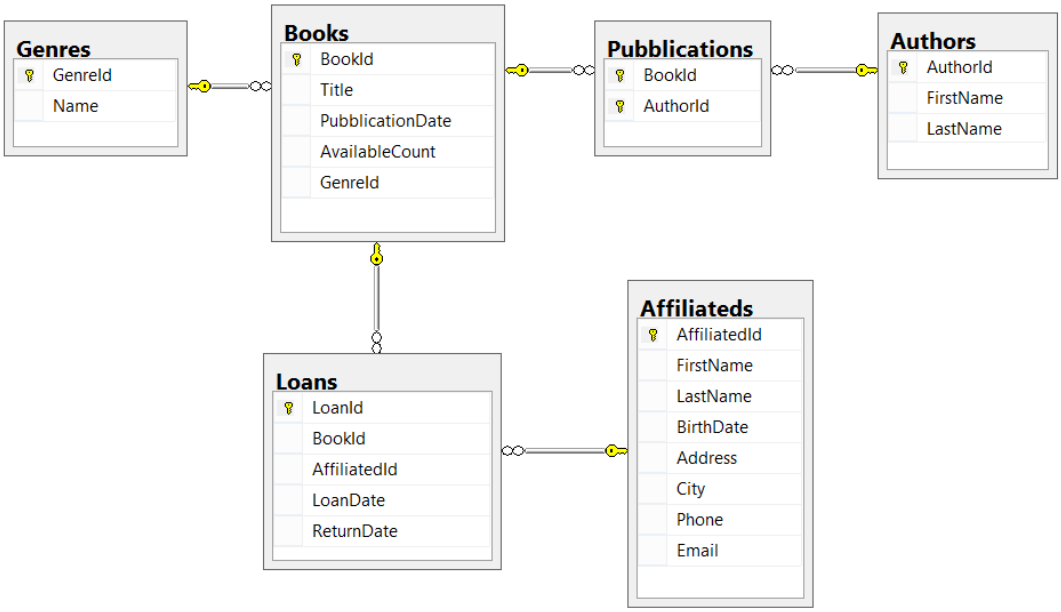
Il codice di configurazione viene collocato nel metodo virtuale `OnModelCreating()`, eseguito automaticamente durante la creazione del *context*, e si avvale dell'oggetto `modelBuilder`. Questo espone il metodo `Entity()`, che consente di specificare l'entità configurare.

Il seguente codice definisce la lunghezza massima della proprietà `Title` della classe `Book`:

```
public class Library: DbContext
{
    ...
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Book>().Property(b => b.Title).HasMaxLength(50);
    }
}
```

4 Programmare con Code-First

Di seguito affronterò alcuni semplici scenari di programmazione C-F. Contestualmente saranno esaminate alcune funzionalità di EF. Negli esempi sarà utilizzato il database **Library**, che ha il seguente schema:



Il database contiene già dei dati, pertanto negli esempi si utilizzerà C-F nella modalità "accesso a un database esistente".

4.1 Domain model utilizzato

Nei vari esempi sarà utilizzato un *domain model* più semplice possibile, che sarà ampliato col procedere del tutorial. Nella sua forma iniziale è il seguente[3]:

```
using System.Data.Entity;
...
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public DateTime PublicationDate { get; set; }
}
...
public class Library : DbContext
{
    public Library(): base("Library")
    {
        Database.SetInitializer<Library>(null);
    }

    public DbSet<Book> Books { get; set; }
}
```

3 Le classi **Book** e **Library** sono presentate assieme, ma risiedono in file distinti.

> **Configurazione progetto**
>
> Il codice presuppone che il file di configurazione **App.Config** definisca una stringa di connessione di nome "**Library**" che referenzi un database esistente.

4.2 Eseguire delle query: LINQ

EF consente di interrogare il database eseguendo delle query sul *domain model* utilizzando **LINQ** (*Language Integrated Query*). LINQ è un linguaggio di interrogazione simile a SQL, che esiste in varie "incarnazioni", in base alla sorgente dati da interrogare:

1. **LINQ to Objects**: interroga oggetti in memoria (vettori, liste, dizionari, etc);
2. **LINQ to XML**: interroga sorgenti in formato XML.
3. **LINQ to Entities**: interroga gli oggetti di un *entity model*.

Una query LINQ inizia specificando un *dbset*. L'esempio che segue visualizza i titoli dei libri:

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

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

La query si legge così: *per ogni libro presente in libri seleziona il titolo.* Il risultato è un elenco di titoli, e cioè di stringhe. EF traduce la query in:

```
SELECT Title FROM Book
```

quindi la invia al database, recupera il risultato e con esso costruisce un modello OO (in questo caso un oggetto di tipo `IQueryable<string>`).

> **Istruzione SQL generata da EF**
>
> Il codice SQL generato è leggermente diverso, anche se identico nel risultato che produce:
>
> ```
> SELECT [Extent1].[Title] AS [Title] FROM [dbo].[Books] AS [Extent1]
> ```

4.2.1 "Esecuzione differita" delle query

Esiste un aspetto fondamentale che riguarda le query LINQ: *queste vengono eseguite soltanto quando si accede al risultato*. Nell'esempio precedente:

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

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

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

"Materializzazione" del risultato di una query

Il risultato di query diventa disponibile anche quando si applica ad essa il metodo `ToList()`, oppure un metodo che "deve" produrre il risultato, come `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);
```

4.2.2 Filtrare e ordinare i dati

Queste operazioni, come in SQL, richiedono l'applicazione delle clausole ***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("{0,-35} {1:d}", b.Title, b.PublicationDate);
}
```

> **LINQ To Entities e DateTime.Parse()**
>
> LINQ to Entities non implementa il metodo `DateTime.Parse()`; per questo motivo è necessario memorizzare la data di riferimento in una variabile e utilizzare quest'ultima nella query.

4.2.3 Uso di extension methods e lambda expressions

Una query può essere scritta in modo più conciso mediante gli *extension methods* e le *lambda expressions*. Riconsideriamo la query:

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

Lo stesso risultato può essere ottenuto così:

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

4.2.4 Caricamento di un'entità in base alla chiave

La classe `DbSet` definisce il metodo `Find()`, in grado di caricare una singola entità in base alla sua chiave:

```
var db = new Library();
var book = db.Books.Find(1);
Console.WriteLine(book.Title);
```

4.3 Mappare le associazioni: *navigation property*

Un delle caratteristiche più importanti di EF è quella di mappare non soltanto le entità ma anche le loro associazioni.

4.3.1 Collection property e Reference property

EF è in grado di stabilire il tipo di associazione esistente tra due entità analizzando le proprietà che nell'una si riferiscono all'altra. Tali proprietà si chiamano ***navigation properties***.

Si consideri il seguente modello:

```
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public DateTime PublicationDate { 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 determinato libro. La proprietà `Books` nella classe `Genre` è una ***collection property***, poiché fornisce l'accesso ai libri di un genere. In base a queste, EF è in grado di stabilire che tra genere e libro esiste una relazione 1↔N. (Sarebbe così anche se fosse definita una soltanto di esse.)

Gli altri tipi di associazione sono:

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

4.3.2 Configurazione dell'associazione: definizione della Foreign-Key

Il precedente modello non mappa correttamente l'associazione tra le tabelle **Genres** e **Books**; in caso di esecuzione del programma sarebbe sollevata un'eccezione.

EF mappa una *reference property* con una FK il cui nome segue il pattern:

<proprietà>_<PK tabella padre>;

Ma la tabella **Books** definisce una chiave esterna di nome **GenreId** e non **Genre_GenreId**.

Si può risolvere il problema definendo un campo che funga da FK nella classe `Book`.

```
public class Book
{
    ...
    public int GenreId { get; set; }        // Mappa implicitamente la FK di Books
    public Genre Genre { get; set; }        // Reference property
}
```

EF è in grado di desumere che `GenreId` corrisponde alla FK omonima nell'associazione tra generi e libri. (La convenzione rispecchia il pattern "<*navigation property*>Id".)

Configurare la proprietà di chiave esterna

Se il nome della FK non rispetta la convenzione usata da EF, è possibile mappare la proprietà mediante l'annotazione `[ForeignKey]`.

Ad esempio, se la tabella **Books** utilizzasse come FK la colonna **FK_Genre**, potremmo decorare la proprietà `Genre` nel seguente modo:

```
public int FK_Genre { get; set; }   // mappa la chiave esterna

[ForeignKey("FK_Genre")]
public Genre Genre { get; set; }
```

4.4 *Independent associations* vs *foreign key associations*

Nel *domain model*, le associazioni 1↔N che non definiscono un campo di chiave esterna sono definite ***independent associations***, mentre quelle che lo definiscono si chiamano ***foreign key associations***. EF è in grado di mappare e gestire correttamente entrambi i tipi di associazione; d'altra parte, alcuni scenari sono molto più semplici da gestire se si utilizza una *foreign key association*. (Le associazione N↔N sono sempre *independent associations*.)

4.5 Collegare le entità in una query: uso di *reference property*

Dopo le precedenti modifiche, è possibile interrogare il modello per conoscere, ad esempio, i libri del genere "Fantascienza":

```
var db = new Library();
var bookList = from book in db.Books
               where book.Genre.Name == "Fantascienza"
               select book;

foreach (var book in bookList)
{
    Console.WriteLine(book.Title);
}
```

EF, traduce la query LINQ in una JOIN tra le tabelle **Genres** e **Books**, simile alla seguente:

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

4.6 Uso delle navigation property: *lazy loading vs eager loading*

Il modello utilizzato finora nasconde un problema, che la precedente query non ha evidenziato. Si consideri il seguente codice, che accede al primo libro e ne visualizza il genere:

```
var db = new Library();
var book = db.Books.First();
Console.WriteLine(book.Genre.Name);        <- solleva una NullReferenceException!
```

La proprietà `Genre` di `book` è nulla e dunque il codice solleva un'eccezione, come se il libro non avesse un genere di appartenenza. Il motivo è che EF, se non diversamente specificato, non carica i dati associati a una determinata entità; in questo caso non carica il genere associato al libro.

Una strada per risolvere questo problema è quella di modificare il modello in modo da indurre EF ad usare la tecnica del ***lazy loading***. (Letteralmente: "caricamento pigro").

4.6.1 Lazy loading

Questo termine designa la capacità di caricare i dati dal database soltanto quando sono utilizzati nel codice. EF è preimpostato per utilizzare questa tecnica, ma perché possa applicarla richiede che le *navigation property* siano definite virtuali:

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

Dopo questa modifica, l'accesso alla proprietà `Genre` di un libro provoca l'esecuzione di una query allo scopo di caricare il genere referenziato, ma soltanto se non è già stato caricato in precedenza:

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

Lazy loading: problemi di performance collegati ai "round-trip" con il database

Il *lazy loading* migliora le prestazioni in alcuni scenari, ma può peggiorarle in altri. Considera il seguente codice, che visualizza tutti i libri e il relativo genere.

```
var db = new Library();
foreach (var book in db.Books)      // un round trip
{
    Console.WriteLine("{0} {1}", book.Title, book.Genre.Name);   // "n" round trip
}
```

Il primo accesso a `db.Books` determina il caricamento dei libri, ma non dei rispettivi generi. Successivamente, per ogni libro viene eseguita una query per caricare il genere di appartenenza. Per caricare i libri e i relativi generi vengono eseguiti "n+1" accessi al database, dove "n" è il numero dei libri.

4.6.2 Eager loading

Il termine *eager loading* designa la tecnica di caricare immediatamente tutti i dati richiesti. Deve essere applicato manualmente, eseguendo il metodo `Include()`.

Il codice precedente può essere riscritto nel seguente modo:

```
var db = new Library();
foreach (var book in db.Books.Include(b=>b.Genre)) // carica i libri e generi
{
    Console.WriteLine("{0} {1}", book.Title, book.Genre.Name);
}
```

`Include()` richiede una *lambda expression,* che, data un'entità, specifichi l'entità collegata da caricare.

Eager loading: problemi di performance collegati alla complessità della query

L'*eager loading* implica l'esecuzione di *join* tra due o più tabelle. Si tratta di query che possono impattare sulle performance, sia per la loro velocità di esecuzione che per la mole di dati caricati.

4.6.3 Explicit loading

L'*explicit loading* unisce le caratteristiche del *lazy loading* e dell'*eager loading*. Come il primo prevede il caricamento separato delle entità correlate. Come il secondo, è il programmatore a stabilire esattamente quando caricarle.

L'*explicit loading* è accessibile mediante il metodo `Entry()` della classe `DbContext`; questo ritorna un oggetto `DbEntityEntry`, che espone una serie di funzionalità sulle entità. Tra queste ci sono i metodi `Reference()` e `Collection()`, che consentono di caricare le entità associate.

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

```
var db = new Library();
Book book = db.Books.First();
db.Entry(book).Reference(b => b.Genre).Load();    // <- il genere viene caricato qui
Console.WriteLine(book.Genre.Name);
```

5 Modifica dei dati

EF gestisce le modifiche alle entità del *domain model* ed è in grado di persisterle nel database. Le funzionalità fornite sono molte:

1. Tracciare le modifiche alle entità, 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 le modifiche concorrenti.

5.1 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. Per questo motivo qualsiasi operazione deve essere eseguita attraverso un *context*, a partire dal caricamento dal database.

Considera il seguente codice, che crea due liste di generi:

```
var list1 = 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 list2 = db.Genres.Take(3).ToList();
```

I generi memorizzati in `list1` sono scollegati dal *context* e, nel loro stato attuale, non possono essere persistiti nel database. Quelli di `list2` sono memorizzati in `db.Genres` e sono tracciati come *unchanged*, poiché rappresentano una copia esatta dei record della tabella **Genres**. Una modifica a quest'ultimi sarebbe tracciata da EF, il quale sarebbe poi in grado di aggiornare il database eseguendo le istruzioni SQL appropriate.

5.2 Stato di un'entità

A seguito di un'operazione di caricamento, creazione, inserimento, modifica o eliminazione, 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.
Detached	L'entità è stata creata da codice e non è ancora tracciata dal *context*. (Si chiama entità **disconnessa**.) Non sarà coinvolta nelle operazioni verso il database.

Stato	Descrizione
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.

Quando viene modificato lo stato di un'entità, si dice che è *registrata* per l'inserimento, la modifica, etc.

5.2.1 Accesso allo stato di un'entità

Tra le funzionalità accessibili mediante `Entry()` c'è la proprietà `State`, che stabilisce lo stato dell'entità. Il seguente codice recupera il primo genere, ottiene l'oggetto `DbEntityEntry` e visualizza lo stato dell'entità (in questo caso *Unchanged*):

```
var db = new Library();
var genre = db.Genres.First();
var entry = db.Entry(genre);
Console.WriteLine(entry.State);
```

5.2.2 Salvataggio delle modifiche

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

```
var db = new Library();
//
// inserisce, modifica, elimina una o più entità
//
db.SaveChanges();
```

Il metodo salva le modifiche in un'unica transazione. Ciò significa che: o tutte le modifiche vengono persistite correttamente, oppure il metodo esegue un ***rollback***, e cioè un ripristino allo stato precedente l'aggiornamento.

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

5.3 Inserimento di un'entità

L'inserimento di una nuova entità avviene aggiungendo un nuovo oggetto al *dbset* corrispondente. Il codice che segue inserisce un nuovo genere:

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

Recupero della PK

Nel precedente esempio, la `SaveChanges()` non solo salva il nuovo genere, ma recupera anche la PK prodotta dal DBMS e la memorizza nel campo corrispondente. (Naturalmente, ciò non vale se la PK non è *identity*; in questo caso, deve essere l'applicazione a stabilire la PK prima di salvare il nuovo oggetto.)

5.4 Modificare un'entità

La modifica di un'entità si ottiene cambiando una o più proprietà e invocando `SaveChanges()`. L'entità deve essere tracciata dal *context*.

Il codice seguente modifica la data di pubblicazione del primo libro in catalogo:

```
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 SQL "UPDATE" simile alla seguente:

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

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

5.5 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 registra come eliminato; infine salva le modifiche:

```
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();            // <- genre è "unchanged"
```

Nota bene: il metodo `Single()` ritorna il primo elemento della query.

5.5.1 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
```

5.5.2 Eliminare un'entità senza caricarla dal database

In alcuni scenari, l'entità da eliminare non è in memoria, ma di essa si conosce già la PK; in questo caso è possibile registrarla per l'eliminazione senza caricarla in memoria. A questo scopo si crea un'entità "fittizia" (*stub entity*), contenente soltanto la chiave; 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 delGenre = new Genre { GenreId = 9 };        //crea un'entità fittizia
db.Genres.Attach(delGenre);                        //attacca l'entità al contesto
db.Genres.Remove(delGenre);                        //registra l'entità per eliminazione
db.SaveChanges();
```

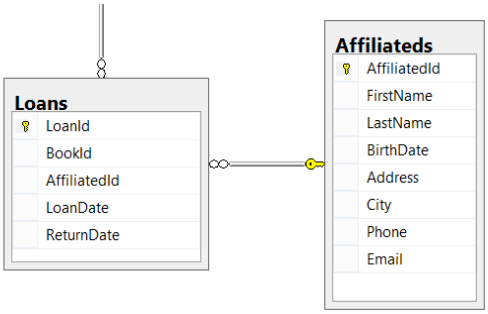
5.5.3 Eliminare un'entità "padre" di un'associazione

Questa scenario si presenta in varie configurazioni:

1. Nel database è definita/non definita una regola di eliminazione a cascata.
2. Nel modello (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 prenderemo in considerazione una situazione tipica, nella quale il database definisce un vincolo di integrità referenziale privo di regola di eliminazione a cascata. In questo caso è responsabilità del programmatore eliminare prima tutte le entità figlie e soltanto successivamente quella padre.

Considera l'associazione **Affiliateds** ↔ **Loans**. Questa è una parte 1→N di una relazione N→N tra tesserati e libri.



Si ipotizzi ora di voler eliminare il tesserato di chiave 3. Innanzitutto, occorre aggiungere al modello le classi che mappano le due tabelle, `Loan` e `Affiliated`. Alla classe `Library` occorre aggiungere i *dbset* `Affiliateds` e `Loans`.

```
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 virtual List<Loan> Loans { get; set; }
}

public class Loan
{
    public int LoanId { get; set; }
    public int AffiliatedId { get; set; }
}
```

> Il modello definisce soltanto i campi necessari per gestire l'eliminazione del tesserato, tra i quali ci sono la proprietà di navigazione `Loans` e il campo di FK `AffiliatedId`. Quest'ultimo, pur non coinvolto nell'operazione, è necessario, altrimenti EF non è in grado di mappare correttamente l'associazione **Affiliateds** ↔ **Loans**.

Segue il codice che elimina il tesserato.

```
var db = new Library();
var affiliated = db.Affiliateds.Find(3);
foreach(var loan in affiliated.Loans.ToList())  // ToList() determina esecuzione query
{
    db.Loans.Remove(loan);                      // rimuove prestiti tesserato
}
db.Affiliateds.Remove(affiliated);
db.SaveChanges();
```

Nota bene:

1. Prima viene caricato il tesserato.
2. Mediante la proprietà di navigazione `Loans` vengono caricati i prestiti del tesserato, i quali vengono registrati per l'eliminazione.
3. Il tesserato viene registrato per l'eliminazione.
4. Vengono salvate le modifiche.

Di seguito, lo script SQL che esegue l'intera operazione, a partire dal caricamento dell'affiliato[4]:

```
SELECT TOP (2) [AffiliatedId], [FirstName], [LastName] FROM Affiliateds
WHERE [AffiliatedId] = @p0
@p0 int, @p0=3
go

SELECT [LoanId], [AffiliatedId], [BookId], [LoanDate], [ReturnDate]FROM [Loans]
WHERE [AffiliatedId] = @EntityKeyValue1
@EntityKeyValue1 int,  @EntityKeyValue1=3
go

DELETE [Loans] WHERE [LoanId] = @0
@0 int, @0=6
```

4 Anche in questo caso il codice è stato “ripulito” allo scopo di facilitarne la lettura.

```
go

DELETE [Loans] WHERE [LoanId] = @0
@0 int, @0=7
go

DELETE [Affiliateds] WHERE [AffiliatedId] = @0
@0 int, @0=3
go
```

EF esegue una DELETE per ogni prestito associato al tesserato e infine una DELETE per eliminare il tesserato.

5.6 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 sulla proprietà di FK dell'entità figlia.

5.6.1 Inserimento di un'entità figlia di un'associazione

Ipotizziamo di voler inserire un nuovo libro, assegnandolo al genere "Fantascienza":

```
var newBook= new Book
{
    Title = "Paratwa",
    PublicationDate = DateTime.Parse("1/1/2003")
};
var db = new Library();
var genre = db.Genres.Where(g => g.Name == "Fantascienza").Single();
genre.Books.Add(newBook);    // aggiunge il libro al genere fantascienza
db.SaveChanges();
```

Nota bene: il libro non è stato inserito in `db.Books`, ma in `genre.Books`. Alternativamente, è possibile sostituire l'istruzione evidenziata con la seguenti:

```
db.Books.Add(newBook);    // aggiunge il libro all'elenco dei libri
newBook.Genre = genre;    // imposta il genere del libro
```

Infine, è possibile impostare direttamente la proprietà di FK (se esiste):

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

In ogni caso, EF inserisce il libro nella tabella **Books**, impostando automaticamente la sua FK al valore corretto.

5.6.2 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.

5.6.3 Rimuovere un'associazione

Rimuovere un'associazione significa eliminare il riferimento all'entità padre. Naturalmente ciò è possibile soltanto se si tratta di un'associazione opzionale e dunque non esiste un vincolo di integrità referenziale. È possibile:

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.

6 Configurazione del *domain model*

Lo schema del database **Library** rispecchia delle convenzioni che semplificano l'operazione di mapping con il *domain model*[5]:

1. Le PK sono campi interi di tipi *identity* e hanno il nome: **<entità>Id**.
2. Le FK sono campi interi e hanno il nome: **<entità associata>Id**.
3. La tabelle hanno il nome al plurale.
4. Non sono definiti vincoli sulle colonne.

Si tratta di una situazione desiderabile, ma che non sempre si presenta nella realtà. In alcuni casi è necessario andare oltre le convenzioni e mappare manualmente il *domain model* con il database.

6.1 Personalizzare la primary key

L'attributo `[Key]` consente di configurare la PK di un'entità.

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

```
using System.ComponentModel.DataAnnotations;       // occorre importare il namespace
...
public class Book
{
    [Key]
    public string ISBN { get; set; }
    public string Title { get; set; }
    ...
}
```

6.1.1 Primary key multi colonna

La tabella **Loans** definisce la PK *identity* **LoanId**; ma altrettanto corretto sarebbe se definisse una PK composta dalle colonne **BookId** e **AffiliatedId**. In questo caso sarebbe necessario configurare manualmente il modello:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema; //occorre importare il namespace

public class Loan
{
    [Key, Column(Order = 0)]
    public int BookId { get; set; }

    [Key, Column(Order = 1)]
    public int AffiliatedId { get; set; }
}
```

5 La tabella **Publications** definisce una PK composta.

6.2 Campi richiesti/opzionali

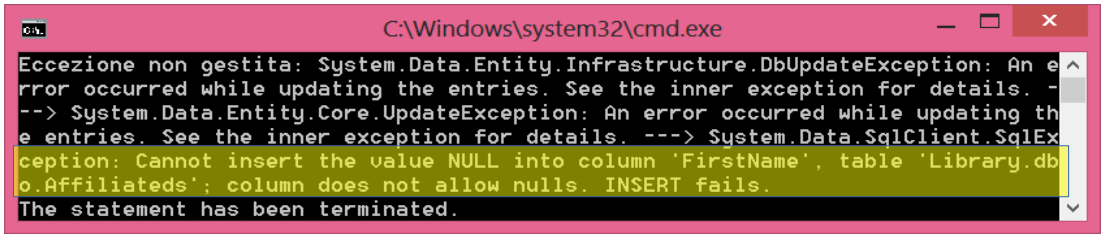
Nel *domain model*, l'opzionalità di un campo segue le regole del linguaggio C#. I *value type* (tipi integrali, `float`, `double`, `decimal`, strutture) sono considerati richiesti. I *reference type* (classi) sono considerati opzionali. Dunque, ad esempio, vale il seguente mapping:

Campo domain model	Colonna database
`public string FirstName {get; set;}`	`--> FirstName NVARCAHR(max)`
`public int BookId {get; set;}`	`--> BookId INT NOT NULL`
`public DateTime LoanDate {get; set;}`	`--> LoanDate DATETIME2 NOT NULL`

Se il database è preesistente, e dunque definisce i propri vincoli, EF tollera una certa incongruenza tra modello e schema, salvo produrre un errore nel momento in cui vengono eseguite operazioni che violino un vincolo. Ad esempio, il seguente codice crea un nuovo tesserato e lo aggiunge al database.

```
var db = new Library();
var a = new Affiliated { LastName = "Giannini" };  // non è impostato FirstName!
db.Affiliateds.Add(a);
db.SaveChanges();         // produce un errore!
```

Il codice produce il seguente errore:



Infatti, nel database sia **FirstName** che **LastName** sono richiesti e dunque non sono accettati record che abbiano un valore nullo nell'uno o nell'altro campo.

6.2.1 Rendere richiesti i reference type: attributo [Required]

L'attributo `[Required]` obbliga EF a verificare che il campo contenga effettivamente un valore. EF valida il vincolo nel momento in cui l'entità viene inserita nel *context*. In questo modo si evita di eseguire operazioni sul database destinate a fallire.

Di seguito utilizzo `[Required]` sui campi `FirstName` e `LastName` della classe `Affiliated`:

```
using System.ComponentModel.DataAnnotations;

public class Affiliated
{
    public int AffiliatedId { get; set; }

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

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

    public virtual List<Loan> Loans { get; set; }
}
```

Dopo questa modifica, il tentativo di aggiungere un tesserato privo di nome e/o cognome provoca un errore.

6.2.2 Rendere opzionali i value type: tipi Nullable<>

È un problema opposto al precedente, ma anche in questo caso si tratta di rendere i vincoli del modello congruenti con quelli del database.

Ad esempio, il seguente codice tenta di aggiungere un nuovo prestito:

```
var db = new Library();
var loan = new Loan                         // non è impostato ReturnDate!
{
    AffiliatedId = 1, BookId = 9,
    LoanDate = DateTime.Parse("1/1/2014")
};
db.Loans.Add(loan);
db.SaveChanges();
```

Il codice è destinato a fallire, poiché il campo `ReturnDate` della variabile `loan` non è stato impostato e dunque assume il valore di default (1/1/0001), che non è compatibile con il tipo **DateTime2** di SQL Server. D'altra parte non è corretto impostare un valore per `ReturnDate`, perché non sarebbe possibile distinguere tra prestiti in corso e prestiti restituiti. La soluzione è rendere `ReturnDate` opzionale, utilizzando un *nullable 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; }      // nota il "?" dopo DateTime
}
```

Dopo questa modifica, l'inserimento di nuovo prestito produce un valore NULL nella colonna **ReturnDate** della tabella **Loans**.

(La stessa modifica deve essere effettuata per il campo `PublicationDate` di `Book`.)

6.3 Mapping di colonne: attributo [Column]

Con l'attributo `[Column]` è possibile personalizzare il mapping tra un campo del modello e una colonna del database. L'attributo consente di specificare il nome e il tipo della colonna di database corrispondente.

Ad esempio, la classe `Affiliated` definisce la proprietà `Email`, che mappa la colonna omonima

della tabella **Loans**. Ipotizziamo che la colonna abbia il nome **Mail** e sia di tipo **NTEXT**, invece che **NVARCHARS**:

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

public class Affiliated
{
    public int AffiliatedId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    ...

    [Column("Mail", TypeName="ntext")]
    public string Email { get; set; }
}
```

6.4 Evitare il mapping di un campo

Per convenzione, EF mappa tutti i campi di una *domain class*, ma non sempre ciò è desiderabile. Una classe potrebbe definire uno o più campi che non hanno una corrispondenza nel database. L'annotazione `[NotMapped]` risolve questo problema, evitando che il campo venga mappato.

Il codice che segue aggiunge la proprietà `FullName` alla classe `Author`, evitando che venga mappata.

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

    [NotMapped]
    public string FullName { get { return LastName + ", " + FirstName;}}
}
```

Attenzione: poiché non corrispondono a colonne nel database, i campi "non mappati" non possono essere usati nelle query. Ad esempio, la seguente operazione produce un'eccezione:

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

Naturalmente è possibile filtrare su `FullName` se la query viene eseguita in memoria, dopo aver caricato gli autori:

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

6.5 Mapping di tabelle: attributo [Table]

L'attributo `[Table]` consente di personalizzare il mapping tra una *domain class* e una tabella. L'attributo è utile quando la denominazione utilizzata nel database non è compatibile con le convenzioni.

Un esempio è rappresentato dall'uso dell'italiano per i nomi di tabella: **Libri**, **Autori**, **Tesserati**, etc. EF non è in grado di tradurre i nomi delle classi `Libro`, `Autore`, etc nei corrispondenti nomi al plurale. Se **Library** fosse denominato in italiano, le *domain classes* potrebbero essere definite nel seguente modo:

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

[Table("Libri")]
public class Libro
{
    public int LibroId { get; set; }
    ...
}

[Table("Autori")]
public class Libro
{
    public int AutoreId { get; set; }
    ...
}
...
```

6.6 Configurare le associazioni N↔N

FF riconosce una relazione N↔N quando due classi si referenziano mediante *collection property*. Ad esempio:

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

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

EF mappa un'associazione N↔N con una tabella contenente le PK delle due entità. Il nome segue il pattern **<classe1><classe2>**, pluralizzato. (**AuthorBooks** o **BookAuthors**, nell'esempio.) Se si desidera modificare il mapping è necessario utilizzare la Fluent API, poiché in questo caso le annotazioni non sono disponibili.

6.6.1 Uso della Fluent API per configurare le relazioni N↔N

In **Library**, la tabella di collegamento tra libri ed autori si chiama **Publications**; poiché il nome non rispetta la convenzione utilizzata da EF, si rende necessario mapparla manualmente.

Nel metodo virtuale `OnModelCreating()` si usa l'oggetto `DbModelBuilder` per configurare ogni elemento dell'associazione:

```
public class Library : DbContext
{
    ...
    protected override void OnModelCreating(DbModelBuilder mb)
    {
        mb.Entity<Author>()                 // un Author
        .HasMany(a => a.Books)              // ha molti Book
        .WithMany(b => b.Authors)           // ognuno dei quali ha molti Author
        .Map(mc =>
        {
            mc.ToTable("Publications");    // Join table: Publications
            mc.MapLeftKey("AuthorId");      // FK Author: AuthorId
            mc.MapRightKey("BookId");       // FK Book  : BookId
        });
    }
}
```

6.6.2 Associazioni N↔N con attributi

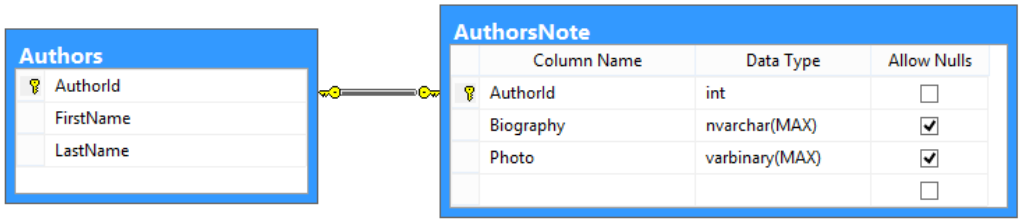
Le associazioni N↔N che definiscono anche degli attributi non possono essere mappate automaticamente da EF. In questo caso è necessario trattare l'associazione come una normale tabella e definire una classe che la mappi.

È questo il caso della classe `Loan`, che rappresenta l'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.

6.7 Configurare associazioni 1↔1

EF riconosce un'associazione 1↔1 quando due classi si referenziano mediante una *reference property*. In questo caso l'uso delle convenzioni non è sufficiente, poiché EF non è in grado di stabilire chi sia l'entità "padre" e chi l'entità "figlia". La soluzione consiste nel configurare la chiave primaria dell'entità figlia come chiave esterna dell'associazione.

Il database **Library** definisce la tabella **AuthorsNote**, in associazione 1↔1 con **Authors**:



Nel modello è necessario aggiungere la classe `AuthorNote` e modificare la classe `Author`.

```
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; }
    public byte[] Photo { get; set; }
    public string Caption { get; set; }
    public Author Author { get; set; }             // ref. property verso Author
}
```

Ebbene, queste modifiche non sono sufficienti, poiché EF non è in grado di stabilire quale delle due classi sia dipendente dall'altra. Né è in grado di interpretare correttamente il ruolo svolto dai campi **AuthorId** nelle due tabelle. Occorre annotare il campo `AuthorId` di `AuthorNote` sia come PK che come chiave esterna, specificando a quale *navigation property* è collegata:

```
[Table("AuthorsNote")]
public class AuthorNote
{
    [Key]
    [ForeignKey("Author")]
    public int AuthorId { get; set; }              // è sia PK che FK

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

Queste annotazioni informano EF che `Author` mappa la tabella padre della associazione, mentre `AuthorNote` mappa la tabella figlia.

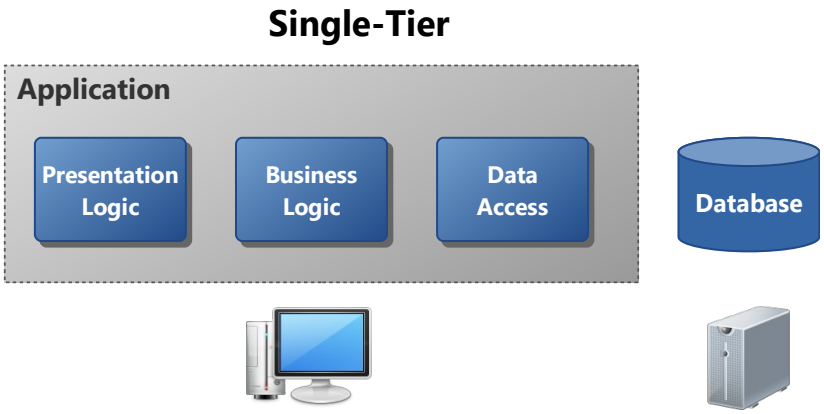
(Nota bene: La classe `AuthorNote` è stata decorata con l'attributo **[Table]**, in modo da mapparla correttamente. È necessario, poiché EF pluralizza il nome delle classi aggiungendo la "s" alla fine, mentre, nel database, la tabella di chiama Author**s**Note.

7 Uso di EF in scenari “N-Tier”

Negli esempi presentati finora le entità (libri, generi, tesserati, etc) sono sempre tracciate dall'oggetto *context*, cosa che rende relativamente semplice gestire gli scenari di modifica dei dati. Si tratta di una situazione tipica delle applicazioni ***single-tier***.

7.1 EF e scenari *single-tier*

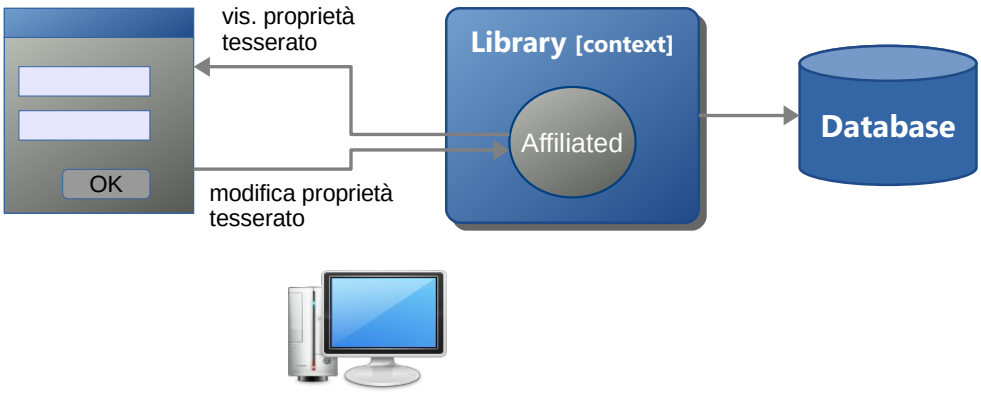
In uno scenario ***single-tier***, l'intero codice applicativo gira all'interno dello stesso processo. La distinzione tra UI (*Presentation logic*) e logica di elaborazione e accesso ai dati, (*Business Logic* + *Data Access*) è puramente logica.



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

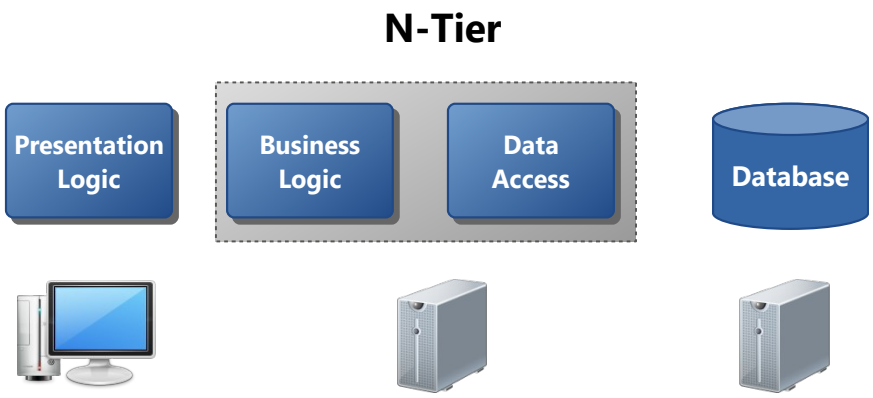
Consideriamo l'ipotesi di modifica dei dati di un tesserato. Il tesserato viene caricato dal database. I suoi dati vengono visualizzati in un *form*, consentendo all'utente di modificarli. Il tesserato, così modificato, viene persistito sul database.



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*

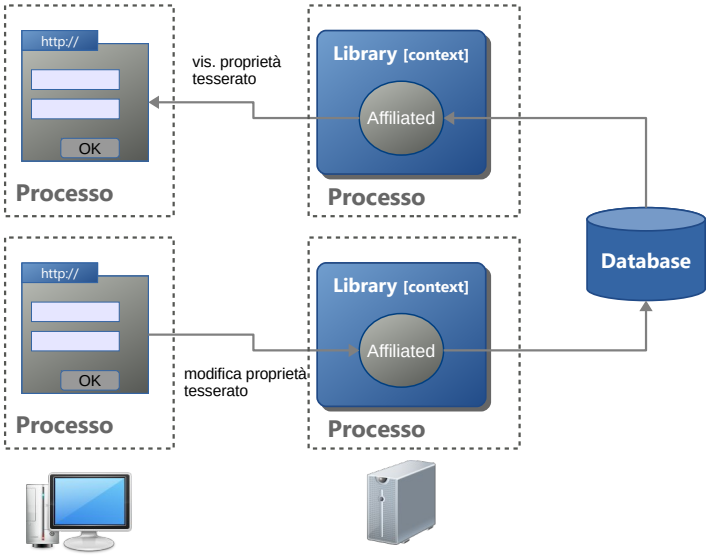
In uno scenario ***n-tier*** l'applicazione è divisa in "strati" che girano su processi diversi, spesso su macchine separate. L'esempio tipico è rappresentato dalle applicazioni web. Il server che ospita l'applicazione (un sito web, ad esempio), elabora le richieste del client, recupera i dati dal database 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

Riconsideriamo l'esempio precedente, 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.

L'oggetto *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.



Nell'esempio delineato, il tesserato modificato rappresenta un'entità **disconnessa**, poiché non è stata precedentemente tracciata dall'oggetto *context* che dovrà elaborarla.

7.3 Programmazione in scenari “disconnessi”

La gestione di entità disconnesse è tipica degli scenari *n-tier*, ma in realtà riguarda anche gli scenari *single-tier*. Alla base c'è la seguente distinzione:

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

Di seguito saranno presentati alcuni esempi di scenari disconnessi. Per mantenere gli esempi di facile comprensione, la “natura disconnessa” sarà simulata: sarà utilizzato 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:

```
public static void AddAuthor(Author au)
{
    var db = new Library();
    db.Authors.Add(au);
    db.SaveChanges();
}
```

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

Più entità associate tra loro rappresentano un **grafo di entità**, nel senso che da una è possibile raggiungere le altre e viceversa. Negli scenari disconnessi questo pone un problema, poiché occorre “istruire” EF sullo stato delle entità associate a quella che si vuole inserire.

Ipotizziamo di inserire un nuovo libro. Nel client, 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 un'entità `Genre` già esistente nel database. Il codice seguente simula l'intero processo

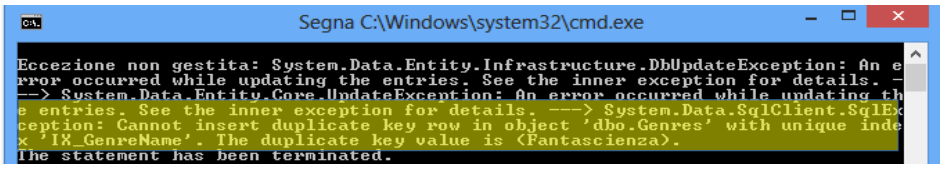
```
public static void TestAddBook()
{
    // SERVER: caricamento del genere 'Fantascienza'
    var db = new Library();
    var genre = db.Genres.Find(1);
    db.Dispose();

    // CLIENT: creazione del nuovo libro e associazione al genere 'Fantascienza'
    var book = new Book { Title = "Cronache della Galassia",
                          Genre = genre, AvailableCount = 1 };

    // SERVER: inserimento del nuovo libro
    AddBook(book);
}
```

```
public static void AddBook(Book book)
{
    var db = new Library();
    db.Books.Add(book);
    db.SaveChanges();
}
```

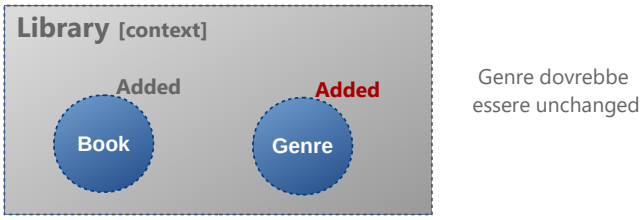
L'esecuzione del precedente codice produce un errore; EF prima di inserire il nuovo libro tenta di inserire il genere, che però esiste già nel database.



Il problema è che l'esecuzione di:

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

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, a meno che queste non siano già tracciate. In questo caso si tratta di un comportamento che complica l'operazione di inserimento.

7.5 Modificare lo stato delle entità: metodo Entity()

Non esiste una strategia generaleper gestire le operazioni su entità disconnesse, ma, qualsiasi approccio si decida di utilizzare, resta necessario modificare esplicitamente lo stato delle entità coinvolte. Il metodo `Entity()` consente sia di ottenere lo stato di un'entità (vedi 5.2.1), che di modificarlo.

7.5.1 Esempio: modificare lo stato da Added a Unchanged

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

```
public static void AddBook(Book book)
{
    var db = new Library();
    db.Books.Add(book);
    db.Entry(book.Genre).State = EntityState.Unchanged;
    db.SaveChanges();
}
```

Alternativamente, è possibile usare il metodo `Attach()`. Questo "attacca" un'entità al contesto, impostandola come *Unchanged*:

```
public static void AddBook(Book book)
{
    var db = new Library();
    db.Books.Add(book);
    db.Genres.Attach(book.Genre);
    db.SaveChanges();
}
```

In entrambi i casi, EF si limita a generare una INSERT per aggiungere il nuovo libro.

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

Nell'inserire il nuovo libro, il metodo `AddBook()` parte dall'assunzione che il genere sia già esistente; non si tratta di una assunzione sempre valida, poiché il client potrebbe aver creato un nuovo libro appartenente a un nuovo genere. Un modo per distinguere tra entità nuove ed esistenti 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.

È possibile usare questa tecnica nel metodo `AddBook()`.

```
public static void AddBook(Book book)
{
    var db = new Library();
    db.Books.Add(book);
    if (b.Genre.GenreId > 0)
        db.Entry(b.Genre).State = EntityState.Unchanged;
    db.SaveChanges();
}
```

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

Dopo questa modifica, il seguente codice:

```
public static void TestAddBook()
{
    // CLIENT: creazione del nuovo libro e genere
    var g = new Genre { Name = "Astrofisica" };
    var b = new Book { Title = "I primi tre minuti", Genre = g, AvailableCount  = 1};

    // SERVER: inserimento nuovo libro
    AddBook(b);
}
```

produce:

1. L'inserimento del nuovo genere.
2. Il recupero della PK del genere appena inserito.
3. La valorizzazione della FK del libro, mediante la PK precedente.
4. L'inserimento del nuovo libro (e recupero della sua PK).

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 e verificare se ritorna un risultato vuoto.

```
public static void AddBook(Book book, string genreName)
{
    var db = new Library();
    var genre = db.Genres.Where(g => g.Name == genreName).SingleOrDefault();
    if (genre == null)
        genre = new Genre { Name = genreName };
    book.Genre = genre;
    db.Books.Add(book);
    db.SaveChanges();
}
```

Ci sono alcune osservazioni da fare:

1. il metodo `SingleOrDefault()` ritorna il primo valore della query, oppure `null` se questa restituisce un *result set* vuoto.
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 metodo che simula il processo di inserimento del nuovo libro.

```
public static void TestAddBook()
{
    // CLIENT: creazione del nuovo libro
    var book = new Book { Title = "I primi tre minuti", AvailableCount  = 1};

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

7.6 Modificare un'entità

Per persistere un'entità come modificata è sufficiente cambiare il suo stato in *Modified*.

```
public static void ModifyBook(Book book)
{
    var db = new Library();
    db.Entry(book).State = EntityState.Modified;
    db.SaveChanges();
}
public static void TestModifyBook()
{
    // SERVER: caricamento del libro
    var db = new Library();
    var book = db.Books.Find(1);
```

```
    db.Dispose();

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

    // SERVER: modifica del libro nel database
    ModifyBook(book);
}
```

La semplice modifica dello stato dell'entità, implica l'aggiornamento di tutte le sue proprietà, come dimostra l'istruzione SQL generata da EF[6]:

```
UPDATE [Books]
SET [Title] = @0, [PublicationDate] = @1, [AvailableCount] = @2, [GenreId] = @3
WHERE ([BookId] = @4)

@0=N'Fondazione', @1='2011-01-01 00:00:00', @2=1, @3=1, @4=1
```

7.6.1 Modificare un'associazione

Come detto in 5.6.2, questo risultato può essere ottenuto in vari modi, agendo sulle *navigation property* 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. Al metodo `ModifyBook()` viene aggiunto un parametro, `genre`, che indica il genere del libro. (Per semplificare il codice, si assume che il genere sia già presente nel database).

```
public static void ModifyBook(Book book, Genre genre)
{
    var db = new Library();
    db.Entry(book).State = EntityState.Modified;
    db.Entry(genre).State = EntityState.Unchanged;
    book.Genre = genre;
    db.SaveChanges();
}

public static void TestModifyBook()
{
    // SERVER: caricamento del libro e del genere
    var db = new Library();
    var book = db.Books.Find(1);
    var genre = db.Genres.Where(g => g.Name == "Fantasy").Single();
    db.Dispose();

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

    // SERVER: modifica del libro nel database
    ModifyBook(book, genre);
}
```

6 Un approccio più sofisticato prevede di tracciare le proprietà modificate, in modo che EF possa generare un'istruzione UPDATE che riduca le modifiche apportate al database.

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 su di essa il metodo `Remove()`; EF risponde registrando l'entità per l'eliminazione, oppure rimuovendola se era nello stato *Added*. Con un'entità "disconnessa" questo 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 `Entity()`.

Il codice che segue elimina un prestito dal database. Nota bene: il codice di test carica il primo prestito per il quale il libro non è stato ancora restituito.

```
public static void DeleteLoan(Loan loan)
{
    var db = new Library();
    db.Entry(loan).State = EntityState.Deleted;
    db.SaveChanges();
}

public static void TestDeleteLoan()
{
    // SERVER: caricamento del prestito (il primo non ancora restituito)
    var db = new Library();
    var loan = db.Loans.Where(lo => lo.ReturnDate == null).First();
    db.Dispose();

    // SERVER: eliminazione del prestito
    DeleteLoan(loan);
}
```

7.8 Migliorare le performance di caricamento: AsNoTracking()

In molte situazioni non è necessario tracciare le entità caricate dal database, perché non sono previste modifiche, oppure, come negli scenari *n-tier*, perché eventuali modifiche sono gestite da un altro *context*. Per evitare che l'entità vengano tracciate, e dunque per migliorare le prestazioni, è possibile invocare il metodo `AsNoTracking()`. Ad esempio:

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

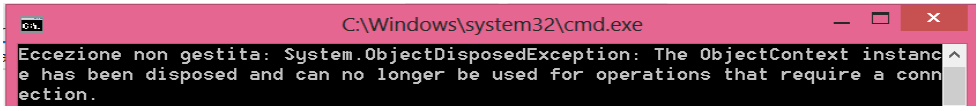
7.9 Scenari disconnessi e Lazy loading

Esiste una problematica riguardante gli scenari disconnessi e la tecnica del *lazy loading*. 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.

```
static void TestDisconnectedLazyLoad()
{
    var db = new Library();
    Book book = db.Books.Find(1);
    db.Dispose();
    ShowBookAuthors(book);
}
static void ShowBookAuthors(Book book)
{
    foreach (var au in book.Authors)              <- qui viene prodotto un errore!
    {
        Console.WriteLine(au.LastName + ", " + au.FirstName);
    }
}
```

Il codice evidenziato produce il seguente errore:



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 `book` a un nuovo *context*, oppure si adotta la tecnica dell'*eager loading* o *dell'explicit loading,* caricando gli autori prima di elaborare il libro.

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*. 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. Esistono comunque degli approcci alternativi.

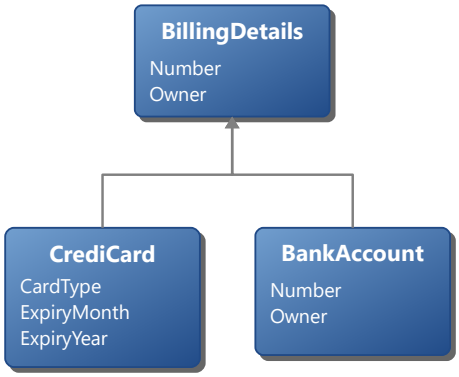
Alcuni sono basati su framework che hanno funzione di tracciare sul client le modifiche e successivamente di applicarle sul server. Un esempio è **WCF Data Services** + protocollo **O-Data**. Questo framework rende trasparente la suddivisione client/server e, di fatto, riduce uno scenario *n-Tier* ad uno *single-Tier*. Un'altra strategia prevede l'implementazione di ***Self Tracking Entities***. Si tratta di 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 affrontare è quella di implementare le associazioni di "generalizzazione" tra le entità. Esistono tre strategie:

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

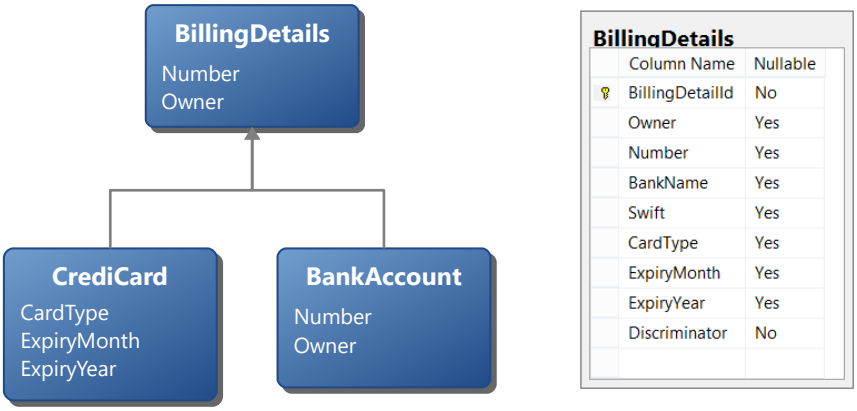
EF consente di configurare il mapping del *domain model* per ognuna di esse. Qui analizzerò soltanto TPH e TPT, prendendo come esempio la seguente gerarchia di entità che definisce i sistemi di pagamento di una fattura.



Alla base del sistema di pagamento c'è un'entità astratta che definisce gli attributi comuni a tutti i sistemi: un numero progressivo (**Number**) e il titolare (**Owner**). Dall'entità **BillingDetails** derivano i sistemi di pagamento effettivi: carta di credito e conto corrente bancario.

8.1 Table per Hierarchy

Con la strategia TPH l'intera gerarchia viene mappata in una sola tabella.



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

Il tipo della colonna discriminatore è in genere intero o stringa; in quest'ultimo caso definisce di solito il nome dell'entità memorizzata nel record: **CreditCard** o **BankAccount**.

Eccetto la PK e la colonna discriminatore, tutte le altre colonne sono opzionali. È un vincolo della strategia TPH e rappresenta il suo difetto principale.

8.2 Convenzioni applicate da Code-First a TPH

EF applica la TPH come strategia di default e dunque non richiede configurazioni.

Segue il *domain model* corrispondente al diagramma della pagina precedente e alla tabella:

```
public abstract class BillingDetail
{
    public int BillingDetailId {get;set;}
    public string Owner {get;set;}
    public string Number {get;set;}
}

public class BankAccount:BillingDetail
{
    public string BankName {get;set;}
    public string Swift {get;set;}
}

public class CreditCard:BillingDetail
{
    public int? CardType {get;set;}
    public string ExpiryMonth {get;set;}
    public string ExpiryYear {get;set;}
}

public class BillingCtx:DbContext
{
    public DbSet<BillingDetail> BillingDetails {get;set;}
}
```

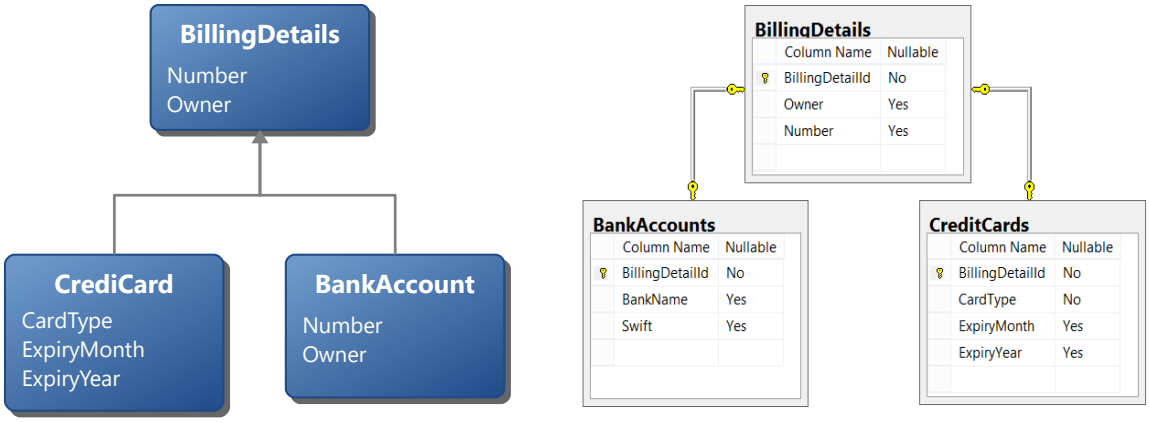
Ci sono alcune osservazioni da fare:

1. Soltanto la classe `BillingDetail` definisce la chiave primaria. Le classi derivate la ereditano.
2. Il *context* definisce il solo *dbset* `BillingDetails`.
3. Il campo `CardType` è di tipo *nullable*. L'unico campo richiesto è la PK.
4. Nel modello non esiste un campo discriminatore, poiché è gestito in modo trasparente da EF.

> **Tipo del campo discriminatore**
>
> Nel database, il tipo del discriminatore può essere configurato mediante *fluent* API.

8.3 Table per Type

Utilizzando TPT la gerarchia viene realizzata mediante associazioni tra tabelle. Ogni classe ha la propria tabella, la quale definisce soltanto le colonne relative ai campi non ereditati.



Si tratta di uno schema normalizzato, dove tra le le tabelle esiste una tipica associazione 1↔1, nella quale la PK della tabella padre diventa PK e FK nelle tabelle figlie.

Quando viene creato un nuovo sistema di pagamento, ad esempio una carta di credito, viene inserito un record sia in **BillingDetails** che in **CreditCards**; le colonne **Owner** e **Number** vengono inserite nella prima tabella, mentre **CardType**, **ExpiryMonth** e **ExpiryYear** vengono inserite nella seconda.

8.4 Convenzioni applicate a TPT

Dato che TPT non è la strategia di default, occorre configurare il modello affinché EF possa utilizzarla: è sufficiente decorare con l'attributo `[Table]` le classi derivate, in modo che possano mappare le rispettive tabelle.

```
public abstract class BillingDetail
{
    public int BillingDetailId {get;set;}
    public string Owner {get;set;}
    public string Number {get;set;}
}

[Table("BankAccounts")]
public class BankAccount:BillingDetail
{
    public string BankName {get;set;}
    public string Swift {get;set;}
}

[Table("CreditCards")]
public class CreditCard:BillingDetail
{
    public int CardType {get;set;}  // <- non è necessario che sia opzionale
    public string ExpiryMonth {get;set;}
```

```
    public string ExpiryYear {get;set;}
}
public class BillingCtx:DbContext
{
    public DbSet<BillingDetail> BillingDetails {get;set;}
}
```

Nota bene: ogni classe è mappata su una tabella e dunque non è più obbligatorio definire i campi come opzionali.

8.5 Gestione “polimorfica” delle entità

A prescindere dalla strategia utilizzata, TPH o TPT, il *context* definisce un unico *dbset* attraverso il quale accedere ai sistemi di pagamento. La collezione `BillingDetails` è polimorfica, poiché il tipo *run-time* delle entità è `BankAccount` o `CreditCard`, in base al record corrispondente memorizzato nel database.

Il seguente codice visualizza l'elenco dei sistemi di pagamento. Per ogni sistema viene visualizzato il nome del proprietario e il tipo (**BankAccount** o **CreditCard**):

```
var bil = new BillingCtx();
foreach (var bd in bil.BillingDetails)
{
    string bdType = (bd is BankAccount)? "BankAccount" : "CreditCard";
    Console.WriteLine(bd.Owner + "  " + bdType);
}
```

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

8.5.1 Query non polimorfiche

La gestione polimorfica è problematica quando si desidera caricare soltanto le entità di un determinato tipo, poiché nel modello non esiste un campo che funga da discriminatore. La soluzione consiste nell'eseguire una query “non polimorfica”, mediante il metodo `OfType<>()`; questo consente di specificare il tipo di entità da caricare.

Il codice seguente visualizza soltanto i sistemi di pagamento con carta di credito.

```
BillingCtx bil = new BillingCtx();
foreach (var cc in bil.BillingDetails.OfType<CreditCard>())
{
    Console.WriteLine(cc.Owner);
}
```

> **Gerarchie di entità con Code-First**
>
> Al link:
>
> 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
>
> c'è una trattazione completa delle strategie di implementazione di una gerarchia di entità.

Appendice I: eseguire il “log” delle istruzioni SQL

Dalla versione 6, EF fornisce un meccanismo per conoscere le istruzioni SQL corrispondenti alle operazioni eseguite dal *context*. A questo scopo esiste il *delegate* `Log` della proprietà `Database` del *context*. Il delegate è di tipo `Action<string>` e dunque deve essere collegato a un metodo `void` che accetta un parametro stringa. Nel seguente codice viene utilizzato il metodo `WriteLog()`:

```
static void Main(string[] args)
{
    Library db = new Librar();
    db.Database.Log = WriteLog;
    ...
}

static void WriteLog(string text)
{
    // text-> istruzione SQL
}
```

8.6 Testo del “log”

Il log riporta l’istruzione SQL eseguita, data e ora di esecuzione, durata e il tipo di risultato (query o no). Ad esempio, il seguente codice:

```
static void Main(string[] args)
{
    Library db = new Library();
    db.Database.Log = WriteLog;
    var books = db.Books.ToList();
    ...
}
```

produce:

```
Opened connection at 02/04/2017 15:25:24 +02:00

SELECT
    [Extent1].[BookId] AS [BookId],
    [Extent1].[Title] AS [Title],
    [Extent1].[AvailableCount] AS [AvailableCount],
    [Extent1].[PublicationDate] AS [PublicationDate],
    [Extent1].[GenreId] AS [GenreId],
    [Extent1].[CoverFileName] AS [CoverFileName]
    FROM [dbo].[Books] AS [Extent1]

-- Executing at 02/04/2017 15:25:24 +02:00
-- Completed in 8 ms with result: SqlDataReader

Closed connection at 02/04/2017 15:25:24 +02:00
```

Nota bene: l’istruzione SQL rispetta una forma particolare a causa dei requisiti imposti al processo di traduzione dal C# al linguaggio SQL.

8.7 Analisi dei costi delle operazioni da EF

Il log è utile per conoscere le operazioni effettivamente eseguite sul DBMS e dunque anche per verificare i vantaggi o svantaggi di una determinata scelta. Ad esempio, il seguente codice, che utilizza l'accesso in modalità *lazy loaded* al genere letterario, produce N+1 query, dove N è il numero dei libri in catalogo:

```
foreach (var b in db.Books)  //-> query su tabella Books
{
    Console.WriteLine("{0} {1}", b.Title, b.Genre.Name); //-> query su tabella Genres
}
```

Segue la prima parte del codice SQL eseguito:

```
Opened connection at 02/04/2017 15:41:08 +02:00

SELECT
    [Extent1].[BookId] AS [BookId],
    [Extent1].[Title] AS [Title],
    [Extent1].[AvailableCount] AS [AvailableCount],
    [Extent1].[PublicationDate] AS [PublicationDate],
    [Extent1].[GenreId] AS [GenreId],
    [Extent1].[CoverFileName] AS [CoverFileName]
    FROM [dbo].[Books] AS [Extent1]

-- Executing at 02/04/2017 15:41:09 +02:00
-- Completed in 8 ms with result: SqlDataReader

SELECT
    [Extent1].[GenreId] AS [GenreId],
    [Extent1].[Name] AS [Name]
    FROM [dbo].[Genres] AS [Extent1]
    WHERE [Extent1].[GenreId] = @EntityKeyValue1

-- EntityKeyValue1: '1' (Type = Int32, IsNullable = false)
-- Executing at 02/04/2017 15:41:09 +02:00
-- Completed in 3 ms with result: SqlDataReader

SELECT
    [Extent1].[GenreId] AS [GenreId],
    [Extent1].[Name] AS [Name]
    FROM [dbo].[Genres] AS [Extent1]
    WHERE [Extent1].[GenreId] = @EntityKeyValue1

-- EntityKeyValue1: '5' (Type = Int32, IsNullable = false)
-- Executing at 02/04/2017 15:41:09 +02:00
-- Completed in 0 ms with result: SqlDataReader
...
```

Appendice II: eseguire direttamente istruzioni SQL

Non esiste una corrispondenza completa tra le operazioni eseguibili da EF e quelle del DBMS; in alcuni scenari può essere necessario, o conveniente, eseguire direttamente istruzioni in linguaggio SQL, per utilizzare funzionalità altrimenti inaccessibili, oppure per aumentare le performance. A questo scopo esiste il metodo `SqlQuery()` della proprietà `Database` del *context*.

Il seguente codice recupera i titoli dei libri:

```
var sql = "SELECT Title FROM Books";
var list= db.Database.SqlQuery<string>(sql);
foreach (var b in list)
{
    Console.WriteLine("{0}", b);
}
```

Nota bene: `SqlQuery()` è un metodo generico; è necessario specificare il tipo dei record restituiti (stringhe, nell'esempio).

8.8 Recupero di record

È necessario che vi sia corrispondenza tra il tipo specificato in `SqlQuery()` e il tipo dai dati restituiti. Questo complica le cose quando il risultato non è un semplice insieme di valori primitivi. Ad esempio, il seguente codice produce un errore:

```
var sql = "SELECT Title, AvailableCount FROM Books";
var books= db.Database.SqlQuery<Book>(sql);
...
```

poiché EF cerca di valorizzare le proprietà dei libri mediante i valori elencati in SELECT, che però specifica soltanto il titolo e disponibilità.

In questo caso, se il risultato della query non corrisponde a nessuna classe *entity*, occorre definire un tipo apposito.

```
class BookInfo
{
    public string Title { get; set; }
    public int AvailableCount { get; set; }
}
...

var sql = "SELECT Title, AvailableCount FROM Books";
var books= db.Database.SqlQuery<BookInfo>(sql);

foreach (var b in books)
{
    Console.WriteLine("{0}", b);
}
```

Appendice III: esempio di query LINQ

L'uso di LINQ nella sua forma naturale (4.2) è particolarmente utile quando occorre eseguire query complesse; in questo caso il codice risulta più leggibile e semplice da scrivere. Consideriamo ad esempio il seguente problema: ottenere l'elenco dei libri sulla base di un tag specificato dall'utente; il tag può essere contenuto nel titolo, nel genere letterario o nel nome dell'autore.

La richiesta coinvolge tre entità; in SQL implicherebbe l'INNER JOIN tra le tabelle **Books**, **Genres**, **Publications** e **Authors**. In LINQ può essere scritta così:

```
var tag = "Fondazione";
...
var books = from b in db.Books
            from a in b.Authors
            where b.Title.Contains(tag) ||
                  b.Genre.Name.Contains(tag) ||
                  a.FirstName.Contains(tag) ||
                  a.LastName.Contains(tag)
            select b;
```

La LINQ query viene tradotta nella seguente istruzione SQL[7]:

```
SELECT
    BookId, Title, AvailableCount, PublicationDate, GenreId, CoverFileName FROM
    Books INNER JOIN
        (SELECT BookId, FirstName, LastName FROM  Publications
         INNER JOIN Authors ON Publications.AuthorId = Authors.AuthorId) AS JOIN1
     ON Books.BookId = JOIN1.BookId

    INNER JOIN Genres AS ON Books.GenreId = Genres.GenreId

    WHERE    (Title LIKE @0) OR (Name LIKE @1) OR
             (FirstName LIKE @2) OR (LastName LIKE @3)

-- @0: '%Fondazione%' (Type = String, Size = 4000)
-- @1: '%Fondazione%' (Type = String, Size = 4000)
-- @2: '%Fondazione%' (Type = String, Size = 4000)
-- @3: '%Fondazione%' (Type = String, Size = 4000)
```

7 Per rendere più leggibile il codice, ho eliminato gli artefatti prodotti da EF, mantenendo comunque la struttura della query.

Appendice IV: stringa di connessione e configurazione

Segue un frammento del file di configurazione di un'applicazione (**App.Config** o **Web.Config**), che mostra come configurare EF e la relativa stringa di connessione:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <section name="entityFramework"
             type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
EntityFramework, Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
requirePermission="false" />
  </configSections>

  <entityFramework>
    <defaultConnectionFactory
type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory, EntityFramework">
      <parameters>
        <parameter value="v11.0" />
      </parameters>
    </defaultConnectionFactory>
    <providers>
      <provider invariantName="System.Data.SqlClient"
type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer" />
    </providers>
  </entityFramework>

  <connectionStrings>
    <add name="nomeConnessione"
         connectionString="Data Source=(localdb)\MSSQLLocalDB; database=nomeDatabase;
                           Integrated Security=true; MultipleActiveResultSets=True"
         providerName="System.Data.SqlClient" />
  </connectionStrings>

</configuration>
```

Le parti in grigio non sono obbligatorie. (Sono necessarie per gestire il database a *design-time*.) Il valore specificato al posto di `nomeConnessione` può essere passato al costruttore base nella definizione del *context*. Ad esempio:

```
public class Library: DbContext
{
    public Library(): base("Library")
    {
        ...
    }
}
```

Appendice V: *namespaces* di Entity Framework

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

System.Data.Entity

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]`.