**Riepilogo della sezione**

Usare un ORM

Un ORM (Object Relational Mapper) è una tecnologia che consente l'accesso ai dati situati in database relazionale. I dati possono essere recuperati con **query fortemente tipizzate**, anziché con query SQL che sono inclini a farci commettere errori di digitazione.

L'ORM si occupa di "tradurre" in SQL le nostre query fortemente tipizzate. Inoltre, legge i risultati trovati dal database e ce li restituisce in forma di oggetti (o grafi di oggetti) grazie a una fase che si chiama **materializzazione**. Un ORM è quindi uno **strato di astrazione** che tende ad ovviare alle rigidità del mondo relazionale e ci permette di sfruttare tutta l'espressività di un linguaggio come il C#. D'altra parte, un ORM ha un **costo prestazionale** proprio perché deve eseguire della **logica di mapping** per "adattare" le peculiarità del mondo relazionale a quelle del mondo a oggetti.

L'ORM che abbiamo valutato si chiama Entity Framework Core ed è stato costruito da Microsoft su ADO.NET, che usa a basso livello per interagire con il database. Ci permette sia di inviare query fortemente tipizzate con LINQ e, all'occorrenza, anche query SQL.

Lambda expression

Scrivere una lambda expression è un modo molto conciso di definire una **funzione anonima** che accetta parametri e restituisce un valore proprio qualsiasi altro metodo C#. Ecco un esempio:

1. //Definisco una lambda expression che riceve due parametri string e restituisce un valore string
2. //L'operatore lambda, cioè il simbolo =>, separa i parametri dal corpo
3. (string firstName, string lastName) => $"{firstName} {lastName}";

Questa scrittura è equivalente alla definizione del seguente metodo (tranne il nome, che nella lambda è assente).

1. private string NomeMetodo(string firstName, string lastName)
2. {
3. return $"{firstName} {lastName}";
4. }

Come dice il nome stesso, una lambda expression è un'*espressione* e perciò può essere **assegnata a una variabile** oppure **passata come argomento** nell'invocare un altro metodo. Ecco un esempio:

1. //Assegno la lambda a una variabile di nome getFullName
2. //Qui i tipi dei parametri non sono neccesari: il compilatore li inferisce dal tipo della variabile
3. Func<string, string, string> getFullName = (firstName, lastName) => $"{firstName} {lastName}";
5. //E poi la passo come argomento nell'invocare un metodo chiamato SortPeople
6. SortPeople(getFullName);

A che scopo passare una lambda expression come argomento? Beh, in questo modo possiamo fornire una nostra logica personalizzata a un altro metodo che invece si occuperà di fare il lavoro "pesante".

Ad esempio, se il metodo si occupa di ordinare un elenco, noi con la lambda expression ci limitiamo ad indicargli il criterio secondo il quale vogliamo che l'elenco sia ordinato (ad esempio: per nome oppure prima per nome e poi per cognome, ecc...).

Le lambda expression possono essere assegnate a variabili di vari tipi:

* Func<T1, T2, ..., TResult> Il tipo Func è usato **quando la lambda expression deve restituire un valore**. L'ultimo argomento di tipo (TResult) indica di che tipo dovrà essere il valore restituito. Tutti i precedenti argomenti di tipo (T1, T2, ...) indicano il numero e il tipo di argomenti che la lambda expression deve accettare;
* Action<T1, T2, ...> Il tipo Action è usato **quando la lambda expression non deve restituire nulla** (equivalente a un metodo void). Gli argomenti di tipo (T1, T2, ...) indicano il numero e il tipo di parametri che la lambda deve accettare.

La sintassi delle lambda expression può variare leggermente:

1. //Le parentesi tonde attorno ai parametri non servono se la lambda expression ne ha esattamente uno
2. Action<string> printName = name => Console.WriteLine(printName);
4. //Se il corpo ha più di una riga di codice, allora servono le graffe e il return
5. Func<DateTime, int, bool> canDrive = (dateOfBirth, legalAge) => {
6. DateTime allowedToDriveFrom = dateOfBirth.AddYears(legalAge);
7. return allowedToDriveFrom <= DateTime.Today;
8. };

Comporre query LINQ

LINQ è una tecnologia che serve a **interrogare elenchi di elementi**, cioè oggetti che implementano l'interfaccia IEnumerable<T>. Per "interrogare" si intende filtrare, ordinare, proiettare e aggregare l'elenco secondo le nostre logiche. A tale scopo fornisce una moltitudine di **extension method**, definiti nel namespace **System.Linq**, che usiamo per comporre delle **query LINQ**.

Ad esempio, l'extension method **Where** si occupa di **filtrare un elenco di elementi** cioè di produrre un nuovo elenco contenente un numero variabile di elementi, da 0 a tutti, in base alla nostra logica che gli passeremo con una lambda expression. Ecco un esempio:

1. //Dato un array di 6 numeri interi
2. var numbers = new int[] { 1, 2, 3, 4, 5, 6 };
4. //Vogliamo ottenere l'elenco dei soli numeri pari
5. //La lambda che forniamo viene chiamata per ciascuno dei numeri in elenco
6. //E restituirà true se il resto della divisione per 2 è 0
7. var evenNumbers = numbers.Where(number => number % 2 == 0);
9. //E li stampiamo con un foreach
10. foreach (var evenNumber in evenNumbers)
11. {
12. Console.WriteLine(evenNumber);
13. }

LINQ possiede numerosi extension method che vedremo nel corso del tempo. Eccone solo alcuni:

* Where produce un nuovo elenco contenente solo gli elementi conformi al nostro criterio di filtro;
* Select proietta gli elementi, cioè produce un nuovo elenco i cuoi elementi sono di altro tipo;
* Sum, Count e Average producono un valore scalare sommando, conteggiando o calcolando la media degli elementi in elenco.

Iniziare con Entity Framework Core

Entity Framework Core, proprio come ADO.NET, è in grado di funzionare con **varie tecnologie database**grazie ai provider realizzati da vari produttori. Per iniziare, dobbiamo quindi installare il pacchetto NuGet idoneo, che ci procurerà il provider per Sqlite. Nel nostro caso, eseguiamo il comando:

1. dotnet add package Microsoft.EntityFrameworkCore.Sqlite

L'elenco completo delle tecnologie database supportate e dei relativi provider si trova a questo indirizzo:  
<https://docs.microsoft.com/it-it/ef/core/providers/>

Creare il modello concettuale

Entity Framework Core ci permette di**lavorare con un modello a oggetti**, anziché direttamente con le tabelle di un database. Quindi il primo passo consiste nel creare le classi del nostro modello a oggetti (anche chiamato "modello concettuale"). Si tratta semplicemente creare delle classi come Course e Lesson che rappresentino le entità che si trovano nel nostro database.

Possiamo seguire due approcci:

* Se seguiamo l'approccio **database-first** vuol dire che abbiamo già creato il nostro database e perciò possiamo generare automaticamente le classi del modello concettuale. Microsoft ha messo a disposizione il comando dotnet ef dbcontext scaffold in grado di fare il *reverse engineering* del database, cioè esaminare le tabelle esistenti e, per ciascuna, generare la rispettiva classe;
* Se invece seguiamo l'approccio **code-first** allora dobbiamo scrivere noi a mano le classi del modello concettuale. In seguito faremo generare lo schema del database a Entity Framework Core grazie alle *migration*, che vedremo più avanti.

È importante notare che l'approccio database-first non è propriamente supportato da Entity Framework Core. Infatti, il tool di *reverse engineering* non è in grado di rilevare le modifiche che apportiamo di volta in volta alla struttura del database e perciò non è in grado di aggiornare le classi nel modello concettuale.

Oltre alle classi del modello concettuale, nel nostro progetto ci sarà una classe che deriva da **DbContext** in cui inseriremo il mapping tra modello a oggetti e modello relazionale.

Mappare il modello concettuale al modello relazionale

Entity Framework Core fa da tramite tra modello a oggetti e modello relazionale. Affinché sia in grado di adempiere a questa responsabilità dobbiamo istruirlo su come ogni classe del modello concettuale rimappa sulla rispettiva tabella (o tabelle) nel database. Il mapping lo indichiamo nel metodo OnModelCreating del DbContext.  
  
Entity Framework Core dispone di un'**interfaccia fluente** che ci permette di mappare:

* **Classi di entità**: sono le classi come Course e Lesson, che hanno una propria identità (cioè una chiave primaria);
* **Owned types**: sono classi come Money, prive di una propria identità e create al solo scopo di tenere coesi alcuni valori (come la valuta e il valore di un importo monetario).
* **Relazioni**: sono i vincoli che legano tra loro due classi di entità. Ad esempio tra Course e Lesson esiste un rapporto di uno-a-molti (per ogni corso esistono più lezioni).

Ecco un esempio di mapping con interfaccia fluente.

1. protected override void OnModelCreating(ModelBuilder modelBuilder)
2. {
3. //Mapping della classe di entità Course
4. modelBuilder.Entity<Course>(entity =>
5. {
6. //Indichiamo tabella e chiave primaria
7. entity.ToTable("Courses");
8. entity.HasKey(course => course.Id);
10. //Mapping di un'owned type
11. entity.OwnsOne(course => course.FullPrice, builder => {
12. builder.Property(money => money.Currency).HasConversion<string>();
13. });
15. //Mapping della relazione uno-a-molti tra Course e Lesson
16. entity.HasMany(course => course.Lessons)
17. .WithOne(lesson => lesson.Course)
18. .HasForeignKey(lesson => lesson.CourseId);
19. });
20. }

Grazie al mapping, possiamo modellare le classi come preferiamo e sfruttare così tutte le potenzialità del linguaggio C# (ereditarietà, proprietà complesse, ecc...). In questo modo non siamo più costretti a seguire una struttura dati rigida come quella del mondo relazionale.

Inoltre, nel metodo OnConfiguring del DbContext dobbiamo ovviamente indicare quale tecnologia database vogliamo usare e qual è la stringa di connessione.

1. protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
2. {
3. if (!optionsBuilder.IsConfigured)
4. {
5. //Indichiamo di voler usare Sqlite e qual è la stringa di connessione
6. optionsBuilder.UseSqlite("Data Source=Data/MyCourse.db");
7. }
8. }

Per finire, creiamo delle proprietà nel DbContext che espongono i DbSet<T>, ovvero gli oggetti a cui rivolgeremo le nostre query LINQ. Ecco dunque l'esempio di un DbContext contenente le parti essenziali.

1. public partial class MyCourseDbContext : DbContext
2. {
3. public MyCourseDbContext(DbContextOptions<MyCourseDbContext> options)
4. : base(options)
5. {
6. }
8. //Proprietà che espongono i DbSet<T>
9. public virtual DbSet<Course> Courses { get; set; }
10. public virtual DbSet<Lesson> Lessons { get; set; }
12. protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
13. {
14. //come già visto nel precedente esempio
15. }
17. protected override void OnModelCreating(ModelBuilder modelBuilder)
18. {
19. //come già visto nel precedente esempio
20. }
21. }

Registrare il DbContext per la dependency injection

Entity Framework Core è **ben integrato con il meccanismo di dependency injection di ASP.NET Core**. I nostri servizi applicativi possono quindi avere una dipendenza dal DbContext, purché sia stato registrato dal metodo ConfigureServices della classe Startup, come si vede in questo esempio.

1. services.AddDbContext<MyCourseDbContext>();

In alternativa possiamo usare il metodo AddDbContextPool<T>, che permetterà di avere **migliori performance** perché le istanze del nostro DbContext saranno create in maniera proattiva, in maniera simile al funzionamento del connection pool di ADO.NET.

1. services.AddDbContextPool<MyCourseDbContext>(optionsBuilder => {
2. //Il contenuto del metodo OnConfiguring va spostato qui
3. //Ricordiamoci di rendere la connection string configurabile dall'esterno dell'applicazione
4. optionsBuilder.UseSqlite("Data Source=Data/MyCourse.db");
5. });

Interrogare il database con query LINQ

A questo punto abbiamo tutto quel che serve per accedere ai dati con Entity Framework Core. Riceviamo un'istanza del nostro DbContext dal costruttore di un servizio applicativo e poi interroghiamo uno dei suoi DbSet<T> con una query LINQ, come la seguente.

1. //Definiamo la query LINQ
2. IQueryable<CourseViewModel> query = db.Courses
3. .AsNoTracking()
4. .Where(course => course.Rating > 4)
5. .Select(course => new CourseViewModel
6. {
7. Id = course.Id,
8. Title = course.Title,
9. //...
10. });
12. //Carichiamo i risultati in una List<T>
13. List<CourseViewModel> results = await query.ToListAsync();

Come si vede qui, abbiamo usato l'extension method ToListAsync per costringere Entity Framework Core ad inviare la query SQL al database e materializzare i risultati in un oggetto List<T>.

Se dobbiamo ottenere un singolo risultato (ad esempio il corso corrispondente a un id) usiamo invece un extension method come FirstAsync o SingleAsync.

È importante notare che la query SQL viene inviata al database il più tardi possibile, cioè in corrispondenza della riga di codice in cui andiamo a leggere i risultati. Questa peculiarità si chiama **deferred execution**.

Altre situazioni in cui Entity Framework Core invia la query SQL sono:

* Quando cicliamo la query LINQ con un foreach;
* Quando usiamo altri extension method di aggregazione come Count, Average, Sum (e loro varianti asincrone).

Per **ottimizzare le prestazioni**, ricordiamoci di:

* Usare l'extension method AsNoTracking se dobbiamo**solo leggere** i risultati. Così evitiamo che Entity Framework Core si metta a tracciare le modifiche alle entità, che in questo caso sarebbe superfluo;
* Fare il mapping tra entità e viewmodel come un'unica espressione all'interno dell'extension method Select (come nell'esempio), altrimenti nella query SQL potrebbero essere coinvolti troppi campi, anche quelli che non ci servono.

Se vogliamo consultare le query SQL inviate da Entity Framework Core, apriamo il riquadro "Debug Console" di Visual Studio Code, dove le vedremo apparire tra i messaggi di logging.