# Migrare da Entity Framework 6 ad Entity Framework Core 5

Quando nel 2016 .NET Core con ASP.NET Core ed Entity Framework Core fecero la loro apparizione, fu subito chiaro che la migrazione di un'applicazione da Framework a Core sarebbe stata un'impresa difficilissima. L'assenza di molte API in .NET Core unito a un cambio di paradigma di sviluppo rendeva quasi impossibile il passaggio. Grazie a .NET Standard 2.0, alla maturazione di .NET Core e soprattutto all'adozione da parte della community, molte librerie sono state adattate per funzionare sia su Framework che Core. Questo ha reso la migrazione di un'applicazione da Framework a Core molto meno complicato. In molti casi, al netto di qualche modifica a tappeto per ASP.NET Core, si può dire che il passaggio fosse quasi indolore.

La situazione tuttavia cambiava radicalmente quando l'applicazione da migrare utilizzava Entity Framework 6. Infatti, Entity Framework 6 conteneva un numero elevato di funzionalità che non erano presenti in Entity Framework Core. Inoltre, alcune funzionalità esistenti in entrambi, erano comunque implementate in modo diverso tra i framework. Per questi motivi, al fine di semplificare la migrazione, si è scelto di modificare Entity Framework 6 per fare in modo che, con piccolissime modifiche al codice dell'applicazione, potesse girare su .NET Core a partire dalla versione 3. Sebbene questo passaggio abbia notevolmente agevolato la migrazione, rimane il problema che Entity Framework 6 è un framework oramai non più evoluto, ma solo manutenuto per bug fix, e che quindi diventerà ben presto inferiore a Entity Framework Core che invece continuerà a crescere con aggiornamenti continui.

Avere un'applicazione dipendente da un prodotto che non verrà evoluto significa condannarsi ad avere un'applicazione che non potrà mai evolvere pienamente e fruire delle ultime novità tecnologiche. Se l'applicazione ha una durata ancora lunga e necessita di essere evoluta, conviene investire nella migrazione a Entity Framework Core. Laddove invece l'applicazione è stabile e non necessità di aggiornamenti o evoluzione, il porting a .NET con l'uso di Entity Framework 6 può essere una valida alternativa. Tuttavia, In questo articolo ci occuperemo della migrazione da Entity Framework 6 a Entity Framework Core 5.

#### Differenze tra Entity Framework 6 ed Entity Framework Core

Da un punto di vista delle API, Entity Framework 6 ed Entity Framework Core condividono molte API. Se guardiamo al codice necessario per scrivere ed aggiornare le entity, potrebbero non esserci addirittura differenze. Tuttavia, al suo interno i framework sono profondamente differenti e in alcuni casi diametralmente opposti.

La principale differenza salta all'occhio già in fase di setup dell'applicazione. Entity Framework 6 può essere importato in un'applicazione in due modi: utilizzando il designer o utilizzando solamente il codice. Entity Framework Core prevede invece il solo utilizzo del codice.

Un'altra differenza tra i due framework è la modalità di recupero della stringa di connessione. Questa differenza è data in parte dalle differenze tra .NET Framework e .NET, ma in parte anche dalle differenze di implementazione della classe **DbContext**.

Anche le modalità di mapping sono cambiate. Infatti, nel caso si fosse usato il mapping via codice anche con Entity Framework 6, le API di mapping sono leggermente cambiate in Entity Framework Core.

Un'altra differenza tra come i due framework è nel modo in cui trattano le relazioni molti a molti. Entity Framework 6 in alcune situazioni costringe a fare dei giri complicati mentre Entity Framework Core permette di ottenere la stessa funzionalità con codice più pulito.

L'ultima importante differenza consiste nel provider LINQ. In Entity Framework Core 5, non si ha ancora una feature parity completa con Entity Framework 6. Tuttavia, la maggior parte delle query viene tradotta al netto di alcuni casi che spesso si riescono a scoprire solo a run time.

Analizziamo nel dettaglio queste differenze e vediamo come possiamo appianarle durante una migrazione.

#### Entity e classe di contesto

La migrazione delle entity e della classe di contesto è probabilmente la parte più complicata dell'intero processo. Esistono due percorsi da intraprendere a seconda che si usi il file edmx in associazione col designer o che si usi solamente il codice.

Quando dobbiamo migrare codice generato dal designer tramite il file edmx, il primo step consiste nel copiare nel progetto .NET 5 i file delle entity. Al netto di un eventuale cambio di namespace, questi file possono essere portati così come sono senza alcuna modifica. Anche il file di contesto può essere portato, ma necessità di due modifiche: la rimozione del costruttore e l'implementazione del metodo **OnModelCreating** con il codice di mapping (di cui parleremo nella prossima sezione).

Quando dobbiamo migrare entity e classe di contesto scritti a mano, il compito è più agevole in quanto possiamo semplicemente copiare nel nuovo progetto le entity e la classe di contesto facendo attenzione a eliminare da quest'ultima eventuali costruttori e ad allineare eventualmente i namespace.

Ovviamente, questi passi rappresentano solo il primo passo del processo. Il secondo è rappresentato dalla scrittura del codice di mapping.

#### Codice di mapping

Ora che nel nostro nuovo progetto abbiamo sia entity che classe di contesto, dobbiamo passare alla scrittura del codice di mapping all'interno del metodo *OnModelCreating*. Nel caso in cui la classe di contesto è stata originariamente generata dal file edmx, questo metodo è contiene una sola riga di codice che lancia un'eccezione che va cancellata e va riempito a mano con tutte le informazioni di mapping necessarie. Nel caso in cui la classe di contesto è stata scritta a mano da noi, questo metodo è già popolato, ma vanno fatte diverse modifiche nel codice in quanto molti metodi di mapping sono diversi in Entity Framework Core. Nella seguente tabella proviamo a mappare tra loro i metodi più comuni.

|  |  |
| --- | --- |
| **Entity Framework 6** | **Entity Framework Core** |
| HasRequired | HasOne(lambda).IsRequired() |
| HasOptional | HasOne(lambda) |
| WithOptional | WithOne(lambda) |
| WithRequired | WithOne(lambda).IsRequired() |
| IsOptional | IsRequired(false) |
| WillCascadeOnDelete | OnDelete |
| Map (relazioni) | Nessun equivalente: usera metodi Has\* and With\* |
| MapToStoredProcedures | Non supportato |
| HasDatabaseGeneratedOption | ValueGenerated\* |
| Map (ereditarietà) | HasDiscriminator (TPH) |

Gli altri metodi presenti in Entity Framework 6 e non menzionati in questa lista sono stati riportati in Entity Framework Core con lo stesso nome. Entity Framework Core inoltre aggiunge molti altri metodi di mapping che rispecchiano le nuove funzionalità.

Una menzione speciale la merita il modo in cui i due framework calcolano di default il nome della tabella su cui una entity mappa. Entity Framework 6 usa un servizio di pluralizzazione basato sul nome della entity mentre Entity Framework Core usa il nome del DbSet che espone la entity o il nome dell'entity stessa se questa non è esposta da un DbSet. Questa differenza può portare a errori di run time in quanto i due framework potrebbero usare nomi diversi per le tabelle. In questi casi possiamo ricorrere al metodo **ToTable** per assegnare il nome corretto quando mappiamo l'entity.

#### EF Core Power Tools

Gli EF Core Power Tools sono degli strumenti che aggiungono a Visual Studio diverse funzionalità per semplificare lo sviluppo con Entity Faramework Core. Tra queste funzionalità, c'è la capacità di generare le entity e la classe di contesto a partire dal database o da un file *.dacpac*. Nelle ultime daily build, è stata aggiunta anche la possibilità di generare il codice partendo da un file ".edmx". Sebbene sia una funzionalità ancora in fase sperimentale, questa potrebbe velocizzare notevolmente i tempi di migrazione soprattutto per quanto riguarda la generazione del codice.

#### Gestire la stringa di connessione

Entity Framework 6 supporta due tipologie di stringa di connessione. La prima viene usata quando si usa l'EDMX e la seconda quando si usa l'approccio code first. Entrambe sono visibili qui sotto.

<add

name="ConnStringEdmx"

connectionString="metadata=res://\*/Model1.csdl|res://\*/Model1.ssdl|res://\*/Model1.msl;

provider=System.Data.SqlClient;provider connection string=&quot;data source=(local);

initial catalog=Northwind;integrated security=True;MultipleActiveResultSets=True;

App=EntityFramework&quot;"

providerName="System.Data.EntityClient" />

<add

name="ConnString"

connectionString="data source=(local);initial catalog=Northwind;integrated security=True;

MultipleActiveResultSets=True;App=EntityFramework"

providerName="System.Data.SqlClient" />

Entity Framework Core accetta solo il secondo tipo di stringa di connessione. Quindi questo è un altro motivo per cui il porting da codice scritto con code first è più semplice.

Un'altra differenza che salta agli occhi guardando le stringhe di connessione è che in .NET Framework queste sono memorizzate all'interno dei file *app.config* o *web.config* in formato XML mentre in .NET queste sono salvate generalmente nel file *appsettings.json* in formato JSON. Questo significa che la stringa di connessione va inserita all'interno di quest'ultimo file e precisamente nella sezione ConnectionStrings.

{

"ConnectionStrings": {

"ConnString": "data source=(local);initial catalog=Northwind;

integrated security=True;MultipleActiveResultSets=True;"

}

}

L'ultimo step consiste nel far leggere la stringa di connessione alla classe di contesto. Questa tecnica cambia in base al fatto che ci troviamo in una app web o in una app windows o console. Nel caso di una app web, possiamo configurare la stringa di connessione usando il metodo **AddDbContext** nel metodo **ConfigureServices** del file *Startup.cs*.

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<NorthwindContext>(

options => options.UseSqlServer("ConnectionStrings:ConnString"));

}

Oltre a questo, dobbiamo aggiungere alla classe di contesto un costruttore che accetta in input un oggetto di tipo **DbContextOptions<T>, dove T è il tipo della classe di context, e che passa questo oggetto al costruttore di base. Il codice è mostrato nel prossimo esempio.**

**public class NorthwindContext : DbContext**

**{**

**public NorthwindContext(DbContextOptions<NorthwindContext> options)**

**: base(options)**

**{**

**}**

**}**

**Se invece ci troviamo in un'applicazione windows o console, possiamo creare nella classe di contesto un costruttore che accetta la stringa di connessione e poi eseguire l'override del metodo OnConfiguring all'interno del quale passiamo la stringa di connessione mentre configuriamo il provider.**

**public class NorthwindContext : DbContext**

**{**

**private readonly string \_connectionString;**

**public NorthwindContext(string connectionString)**

**{**

**\_connectionString = connectionString;**

**}**

**protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)**

**{**

**optionsBuilder.UseSqlServer(\_connectionString);**

**}**

**}**

**Arrivati a questo punto della migrazione, il codice della classe di contesto è quasi pronta. L'ultima cosa a cui prestare attenzione è la relazione many-to-many.**

#### **Gestire le relazioni many-to-many**

**Come evidenziato nella tabella che mostra le differenze tra i metodi di mapping, il many-to-many è un tipo di relazione che ha subito modifiche tra i due framework. Nel seguente codice, possiamo vedere le differenze.**

***//EF6***

**protected override void OnModelCreating(DbModelBuilder modelBuilder)**

**{**

**modelBuilder.Entity<Student>()**

**.HasMany<Course>(s => s.Courses)**

**.WithMany(c => c.Students)**

**.Map(cs =>**

**{**

**cs.MapLeftKey("StudentRefId");**

**cs.MapRightKey("CourseRefId");**

**cs.ToTable("StudentCourse");**

**});**

**}**

***//EF Core***

**protected override void OnModelCreating(DbModelBuilder modelBuilder)**

**{**

**modelBuilder**

**.Entity<Course>()**

**.HasMany(p => p.Students)**

**.WithMany(p => p.Courses)**

**.UsingEntity(j => j.ToTable("StudentCourse"));**

**}**

**La prima parte di mapping è identica perché entrambi i framework espongono i metodi HasMany e WithMany. La differenza emerge nel metodo che va a stabilire i dati della join table. Entity Framework 6 usa il metodo Map per specificare i campi che fanno parte della join table e il nome della tabella, mentre Entity Framework Core usa il metodo UsingEntity al quale basta il nome della join table; il resto viene definito tramite convenzioni dal framework sebbene possiamo comunque personalizzare anche qui il nome dei campi.**

**Sebbene il porting delle relazioni many-to-many non sia difficoltoso, è giusto sottolineare che Entity Framework Core offre moltissime funzionalità in più rispetto al suo predecessore. È bene quindi verificare le funzionalità offerte da Entity Framework Core prima di portare il codice così da capire quali possono servire meglio i nostri scopi.**

**Arrivati a questo punto, il codice delle entiyt e della classe di contesto è completamente migrato. Possiamo quindi passare al codice necessario per scrivere e leggere dati.**

#### Query e aggiornamenti

Dal punto di vista del codice necessario per scrivere e aggiornare i dati, fortunatamente i due framework sono simili.

Per quanto riguarda le query, LINQ è rimasto identico in entrambi i framework quindi il codice da scrivere per eseguire query è uguale. A cambiare tuttavia è stato il provider che trasforma le query LINQ in SQL. Questo provider è stato riscritto due volte in Entity Framework Core e sebbene sia abbastanza maturo, ci sono ancora alcuni casi in cui una query, che viene perfettamente tradotta in Entity Framework 6, causa invece un'eccezione in Entity Framework Core. Scoprire questi casi in fase di compilazione è impossibile quindi le uniche strade percorribili per intercettarli sono quella di avere o una buona batteria di test oppure quella di aver uno o più tester che fanno un giro completo dell'applicazione.

Anche in questo caso, Entity Framework Core offre funzionalità che Entity Framework 6 non offre (Filtro nelle Include, AsSplitQuery e così via). Durante la migrazione del codice è bene prestare attenzione a queste nuove funzionalità che possono semplificare il codice.

Per quanto riguarda la scrittura, le API sono fondamentalmente le stesse inquanto abbiamo a disposizione **Attach**, **Add**, **Remove** e **SaveChanges(Async)**. Va fatta però attenzione a una differenza fondamentale nei metodi *Add* e *Attach* quando passiamo in input un oggetto che contiene altri oggetti collegati (grafo).

Entity Framework 6 naviga tutti gli oggetti e se questi non sono già collegati al contesto, vengono messi rispettivamente in stato di **Added** o **Unchanged**.

Entity Framework Core invece effettua un ragionamento diverso:

* se la primary key dell'oggetto collegato è generata dal database e il suo valore è quello di default per il tipo (0 per int, null per stringhe, e così via) allora l'entity è messa in stato di Added;
* se la primary key dell'oggetto collegato è generata dal database e il suo valore è diverso da quello di default per il tipo allora l'entity è messa in stato di *Unchanged*;
* Negli altri casi l'entity collegata viene messa in stato di *Added* o *Unchanged* a seconda del metodo usato.

Queste differenze potrebbero portare a cambiamenti in come un grafo di entity viene persistito sul database e anche qui, come per le query, è impossibile intercettare gli errori a design time. L'unico metodo per rilevare eccezioni o comportamenti anomali è con test automatizzati o manuali.

##### **Utilizzo dei proxy**

Entrambi i framework permettono l'utilizzo di classi peroxy per il tracciamento delle modifiche ai dati. Entity Framework 6 offre nativamente questa funzionalità. Basta marcare le prorietà come virtual e abilitare i proxy in configurazione e il gioco è fatto. Entity Framework Core obbliga a scrivere le entity e ad abilitare i proxy in configurazione esattamente come il suo predecessore, ma in aggiunta necessita di un package a parte (**Microsoft.EntityFrameworkCore.Proxies**) a simboleggiare il fatto che seppur presente, questa funzionalità è fortemente sconsigliata e da usare solo nei casi di porting.

#### Conclusioni

Eseguire il porting del codice da Entity Framework 6 a Entity Framework Core è probabilmente l'operazione più costosa dell'intero porting da .NET Framework a .NET. Nella mia personale esperienza ho passato molto più tempo a ricontrollare e verificare query, scritture ed errori di mapping che ha rivisitare controller o view o API.

Questo articolo mostra un pattern di migrazione ben definito, ma ovviamente non può prendere in considerazione tutte le possibili problematiche che si incontrano. Tuttavia, seguendo queste linee guida, si può arrivare a una buona base di partenza per continuare poi il lavoro.