## **Introduce**

### ***What is EF Core?***

EF Core hay Entity Framework Core là một khung đối tượng / ánh xạ quan hệ (ORM). Đây là một cải tiến của ADO.NET cung cấp cho các nhà phát triển một cơ chế tự động để truy cập và lưu trữ dữ liệu trong cơ sở dữ liệu.

Nó là mã nguồn mở, nhẹ, có thể mở rộng và là phiên bản đa nền tảng của công nghệ truy cập dữ liệu Entity Framework.

EF Core được thiết kế để sử dụng với các ứng dụng .NET Core. Tuy nhiên, nó cũng có thể được sử dụng với các ứng dụng dựa trên .NET 4.5+ framework tiêu chuẩn.

### ***Install Entity Framework Core***

Entity Framework Core có thể được sử dụng với các ứng dụng dựa trên .NET Core hoặc .NET Framework 4.6. Ở phần này, bạn sẽ tìm hiểu cách cài đặt và sử dụng Entity Framework Core 2.0 trong các ứng dụng .NET Core bằng Visual Studio 2017.

EF Core không phải là một phần của .NET Core và .NET Framework tiêu chuẩn. Nó là một gói NuGet. Bạn cần cài đặt các gói NuGet cho hai thành phần sau đây để sử dụng EF Core trong ứng dụng của bạn:

1. EF Core DB provider.
2. EF Core tools.

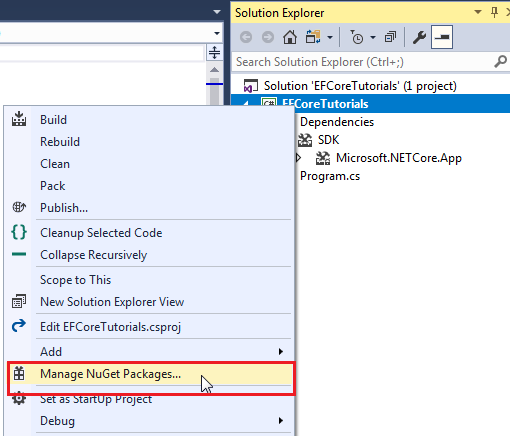
#### **Install EF Core DB provider**

EF Core cho phép chúng ta truy cập cơ sở dữ liệu thông qua Model.

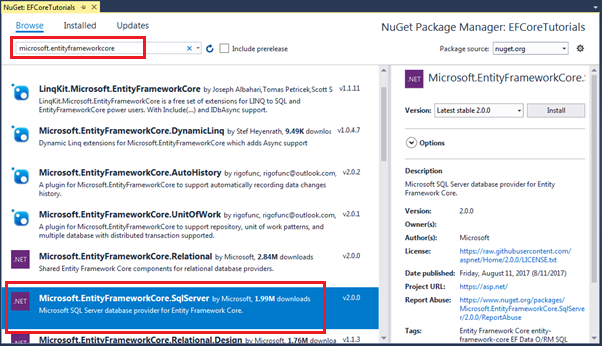
Mỗi database khác nhau lại có EF Core DB providerkhác nhau. Các provider này có sẵn dưới dạng các package NuGet.

Trước tiên, chúng ta cần cài đặt gói NuGet EF Core DB providermà chúng ta muốn truy cập. Ở đây, chúng tôi muốn truy cập cơ sở dữ liệu MS SQL Server, vì vậy chúng tôi cần cài đặt gói NuGet **Microsoft.EntityFrameworkCore.SqlServer**.

Để cài đặt gói NuGet của DB provider, bạn nhấp chuột phải vào project trên Solution Explorer trong Visual Studio và chọn Manage NuGet Packages.. (hoặc chọn trên menu: Tools -> NuGet Package Manager -> Manage NuGet Packages For Solution).

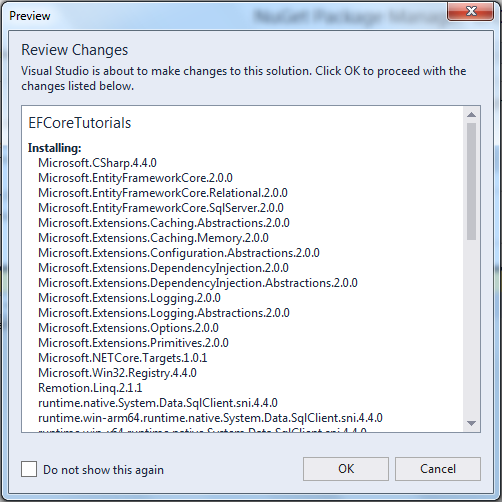


Sau khi giao diện trình quản lý NuGet package sẽ xuất hiện, hãy nhấp vào tab Browse hoặc tab Updates và tìm kiếm **Microsoft.entityframeworkcore** trong hộp tìm kiếm ở góc trên cùng bên trái, như hình ảnh bên dưới.

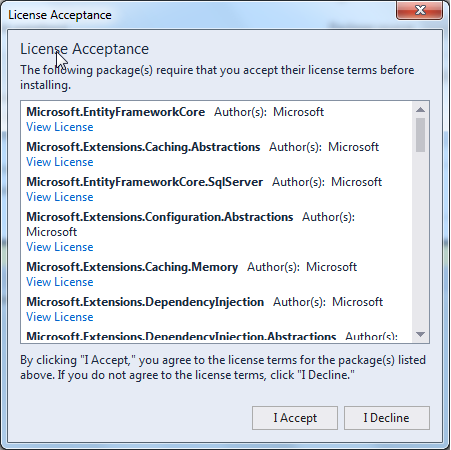


Chọn DB provider bạn muốn truy cập. Trong trường hợp này, chọn **Microsoft.EntityFrameworkCore.SqlServer** cho MS SQL Server như hình trên. (đảm bảo rằng nó có logo .NET và tác giả là Microsoft). Nhấp vào Install để bắt đầu cài đặt.

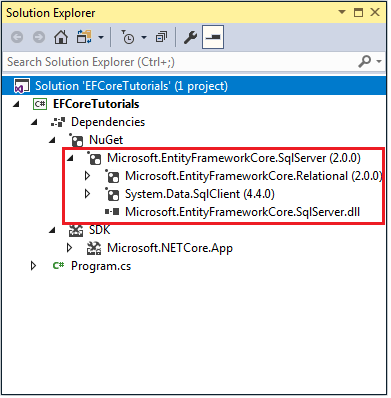
Cửa sổ xem trước hiển thị danh sách các package sẽ cài đặt trong project của bạn. Xem lại các thay đổi và nhấp vào OK.



Cuối cùng, chấp nhận các điều khoản cấp phép liên quan đến các gói sẽ được cài đặt.



Sau khi package **Microsoft.EntityFrameworkCore.SqlServer** install xong, ta có thể kiểm tra nó trong Dependencies -> NuGet, như hình ảnh bên dưới.



**Lưu ý**: NuGet package của provider cũng đã cài đặt các package phụ thuộc khác như **Microsoft.EntityFrameworkCore.Relational** và **System.Data.SqlClient**.

**Ngoài ra,** bạn cũng có thể cài đặt NuGet package của provider bằng Package Manager Console. Chuyển đến Tools -> NuGet Package Manager -> Package Manager Console và thực hiện lệnh sau để cài đặt SQL Server provider package:

PM> Install-Package Microsoft.EntityFrameworkCore.SqlServer

#### **Install EF Core tools**

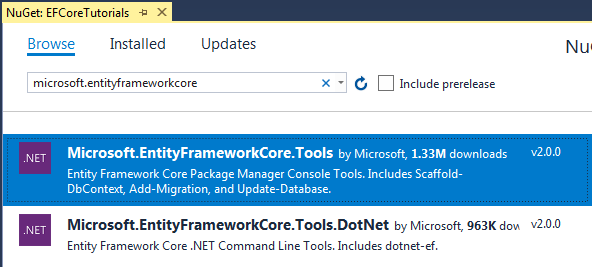
Cùng với DB provider package, bạn cũng cần cài đặt EF tools để thực thi các lệnh EF Core.

Điều này giúp dễ dàng thực hiện một số nhiệm vụ liên quan đến EF Core trong project của bạn tại thời điểm design, chẳng hạn như migrations, scaffolding, v.v.

EF tools có sẵn dưới dạng các NuGet package. Bạn có thể cài đặt NuGet package cho EF tools tùy thuộc vào nơi bạn muốn thực thi các lệnh: sử dụng Package Manager Console (phiên bản PowerShell của các lệnh EF Core) hoặc sử dụng dotnet CLI.

##### ***Cài đặt công cụ EF Core cho PMC***

Để thực thi các lệnh EF Core từ Package Manager Console, hãy tìm kiếm gói **Microsoft.EntityFrameworkCore.Tools** từ NuGet UI và cài đặt nó như hình ảnh bên dưới.



Điều này sẽ cho phép bạn thực thi các lệnh EF Core như scaffolding, migrations,.v.v… trực tiếp từ Package Manager Console (PMC) trong Visual Studio.

##### ***Cài đặt công cụ EF Core cho dotnet CLI***

Nếu bạn muốn thực thi các lệnh EF Core từ CLI (Giao diện dòng lệnh) của .NET Core, trước tiên hãy cài đặt NuGet package **Microsoft.EntityFrameworkCore.Tools.DotNet** bằng NuGet UI.

Sau khi cài đặt gói **Microsoft.EntityFrameworkCore.Tools.DotNet**, chỉnh sửa file .csproj bằng cách nhấp chuột phải vào dự án trong Solution Explorer và chọn Edit <projectname>.csproj. Thêm node **<DotNetCliToolReference>** như ví dụ dưới đây. Đây là một bước bổ sung mà bạn cần thực hiện để thực thi các lệnh EF Core 2.0 từ dotnet CLI trong Visual Studio 2017.

**<Project Sdk="Microsoft.NET.Sdk">**

**<PropertyGroup>**

**<OutputType>Exe</OutputType>**

**<TargetFramework>netcoreapp2.0</TargetFramework>**

**</PropertyGroup>**

**<ItemGroup>**

**<PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="2.0.0" />**

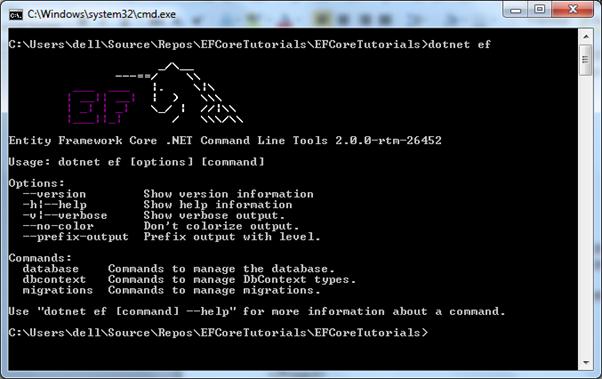
**<PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="2.0.0" />**

**<DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />**

**</ItemGroup>**

**</Project>**

Bây giờ, hãy mở dấu nhắc lệnh (hoặc thiết bị đầu cuối như PowerShell chẳng hạn) từ thư mục gốc của dự án của bạn và thực hiện các lệnh EF Core từ CLI bắt đầu bằng lệnh **dotnet ef**, như hình ảnh bên dưới.

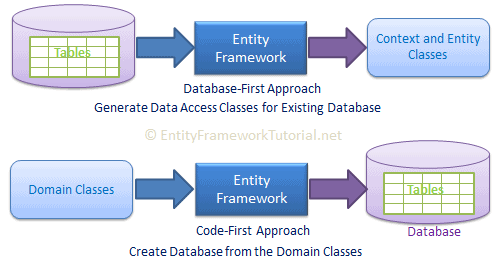


### ***Development Approaches***

EF Core hỗ trợ hai cách tiếp cận phát triển: Code First và Database First. EF Core chủ yếu nhắm vào cách tiếp cận Code First và cung cấp ít hỗ trợ cho cách tiếp cận Database First vì trình thiết kế trực quan hoặc trình hướng dẫn cho mô hình DB không được hỗ trợ kể từ EF Core 2.0.

Với Code First, EF Core API tạo cơ sở dữ liệu và các bảng bằng cách sử dụng chuyển đổi (migration) dựa trên các quy ước và cấu hình được cung cấp trong các Entity class của bạn. Cách tiếp cận này rất hữu ích trong thiết kế hướng miền (Domain Driven Design - DDD).

Còn với Database First, EF Core API tạo các Entity class và Context dựa trên cơ sở dữ liệu hiện có của bạn bằng cách sử dụng các lệnh của EF Core. Điều này được hỗ trợ hạn chế trong EF Core vì nó không hỗ trợ trình thiết kế trực quan (designer) hoặc trình hướng dẫn (wizard).

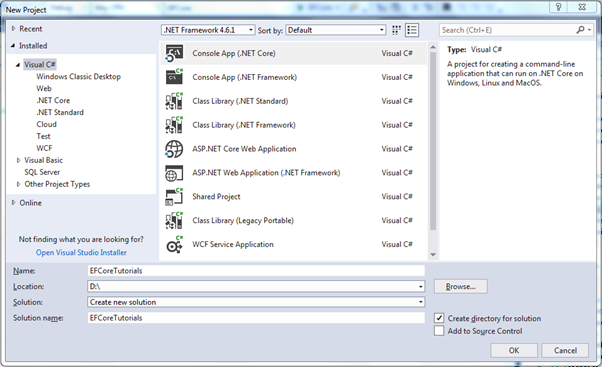


#### **Code First**

Ở hướng dẫn này, bạn sẽ tìm hiểu từng bước cách sử dụng Entity Framework Core theo cách Code First. Để minh họa, tôi sẽ tạo một ứng dụng .NET Core Console bằng Visual Studio 2017 (hoặc mới hơn).

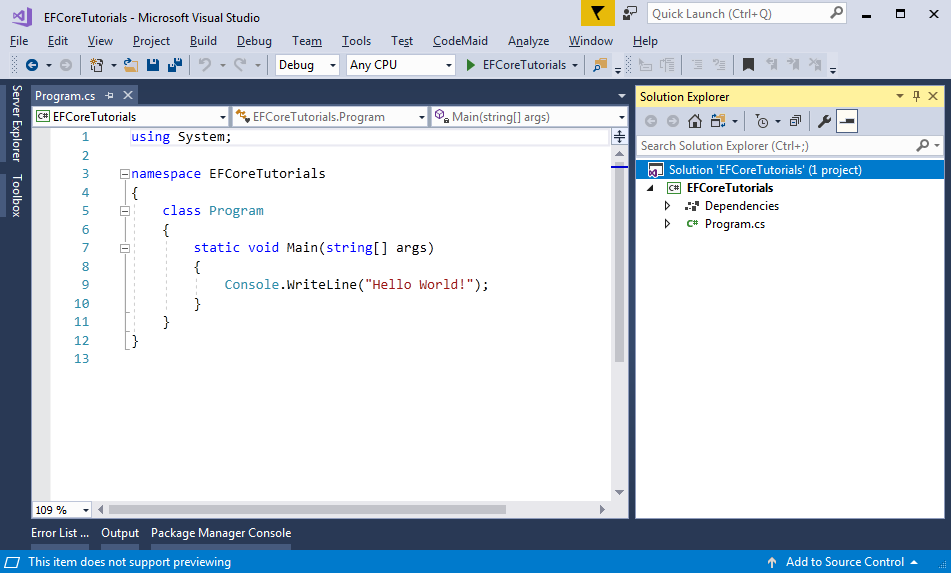
Ứng dụng .NET Core Console có thể được tạo bằng Visual Studio 2017 hoặc Giao diện dòng lệnh (CLI) cho .NET Core. Ở đây chúng tôi sẽ sử dụng Visual Studio 2017.

Để tạo ứng dụng .NET Core Console, hãy mở Visual Studio 2017 và chọn trên menu: File -> New -> Project.. Điều này sẽ mở cửa sổ bật lên New Project như hình ảnh bên dưới.



Trong cửa sổ New Project, mở rộng **Installed -> Visual C#** ở khung bên trái và chọn mẫu **Console App (.NET Core)** trong ngăn giữa.

Nhập tên dự án, vị trí đặt dự án và nhấp vào nút **OK** để tạo ứng dụng bảng điều khiển, như hình ảnh bên dưới.



Bây giờ, chúng ta cần cài đặt EF Core trong ứng dụng của mình bằng Package Manager Console. Chọn trên menu: Tools -> NuGet Package Manager -> Package Manager Console và thực hiện lệnh sau để cài đặt gói nhà cung cấp SQL Server:

PM> Install-Package Microsoft.EntityFrameworkCore.SqlServer

##### *Tạo Entity Model*

Entity Framework cần có một Entity Data Model để giao tiếp với Database. Nó xây dựng một Model dựa trên các entity class của bạn bằng cách sử dụng cấu hình Data Annotation và Fluent API.

Mô hình EF bao gồm ba phần: model khái niệm, model lưu trữ và mapping giữa model khái niệm và model lưu trữ.

Theo cách tiếp cận Code First, EF xây dựng model khái niệm dựa trên các entity class của bạn, lớp Context và các cấu hình.

EF Core xây dựng model lưu trữ và mapping dựa trên Db provider bạn sử dụng. Ví dụ, model lưu trữ của SQL Server sẽ khác với DB2.

EF sử dụng model này cho các action CRUD (Create, Read, Update, Delete) cho cơ sở dữ liệu.

##### *Tạo Migration trong EF Core*

EF Core có các lệnh Migration khác nhau để tạo hoặc cập nhật cơ sở dữ liệu dựa trên model.

Tại thời điểm này chưa có cơ sở dữ liệu SchoolDB. Vì vậy, chúng ta cần tạo database từ model (entity và Context) bằng cách thê migration.

Chúng ta có thể thực thi lệnh migration bằng cách sử dụng Package Manger Console hoặc dotnet CLI (giao diện dòng lệnh).

Trong Visual Studio, mở NuGet Package Manager Console từ Tools -> NuGet Package Manager -> Package Manager Console và nhập lệnh sau:

PM> add-migration CreateSchoolDB

Nếu bạn sử dụng dotnet CLI, hãy nhập lệnh sau.

> dotnet ef migrations add CreateSchoolDB

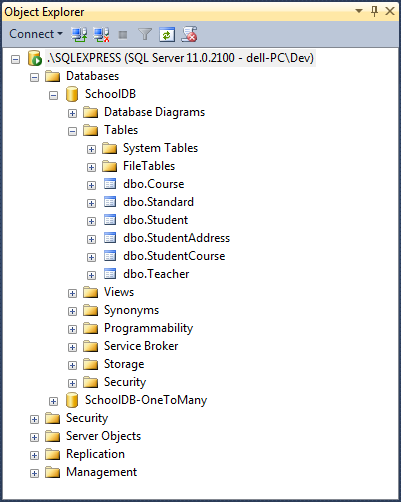
#### **Database First**

Ở phần này bạn sẽ tìm hiểu cách tạo lớp Context và các lớp thực thể cho cơ sở dữ liệu có sẵn trong Entity Framework Core. Tạo các lớp thực thể và Context cho cơ sở dữ liệu có sẵn được gọi là phương pháp tiếp cận Database First.

EF Core không hỗ trợ trình thiết kế trực quan cho mô hình DB và trình hướng dẫn để tạo các thực thể và các lớp Context tương tự như EF 6.

Vì vậy, chúng ta cần thực hiện kỹ thuật đảo ngược bằng cách sử dụng lệnh Scaffold-DbContext. Lệnh đảo ngược này tạo ra các lớp thực thể và Context (bằng cách kế thừa từ lớp DbContext) dựa trên lược đồ của cơ sở dữ liệu hiện có.

Chúng ta sẽ tạo các lớp thực thể và Context cho cơ sở dữ liệu SchoolDB sau trong MS SQL Server như hình ảnh bên dưới.



##### ***Lệnh Scaffold-DbContext***

Sử dụng lệnh Scaffold-DbContext để tạo một mô hình dựa trên cơ sở dữ liệu hiện tại của bạn. Các tham số sau có thể được chỉ định cho Scaffold-DbContext trong Package Manager Console:

Scaffold-DbContext [-Connection] [-Provider] [-OutputDir] [-Context] [-Schemas>] [-Tables>]

[-DataAnnotations] [-Force] [-Project] [-StartupProject] [<CommonParameters>]

Trong Visual Studio, chọn menu Tools -> NuGet Package Manger -> Package Manger Console và chạy lệnh sau:

PM> Scaffold-DbContext "Server=.\SQLExpress;Database=SchoolDB;Trusted\_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models

Trong lệnh trên, tham số đầu tiên là một chuỗi kết nối bao gồm ba phần: Máy chủ DB, tên cơ sở dữ liệu và thông tin bảo mật.

Ở đây, Server=.\SQLExpress; đề cập đến máy chủ cơ sở dữ liệu SQLEXPRESS cục bộ. Database=SchoolDB; chỉ định tên cơ sở dữ liệu "SchoolDB" mà chúng ta sẽ tạo các lớp. Trusted\_Connection=True; chỉ định xác thực Windows. Nó sẽ sử dụng thông tin đăng nhập Windows để kết nối với SQL Server.

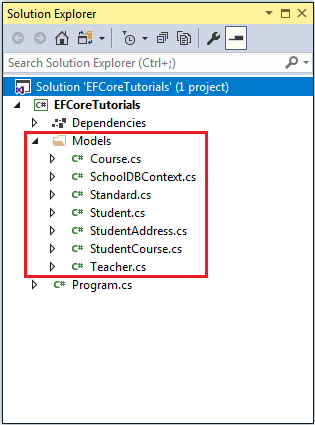
Tham số thứ hai là tên nhà cung cấp. Chúng tôi sử dụng nhà cung cấp cho SQL Server, vì vậy nó là Microsoft.EntityFrameworkCore.SqlServer.

Tham số -OutputDir mô tả thư mục nơi mà chúng tôi muốn tạo ra tất cả các lớp thực thể, đó là thư mục Models.

Sử dụng lệnh sau để xem chi tiết về lệnh Scaffold-DbContext:

PM> get-help scaffold-dbcontext –detailed

Lệnh trên Scaffold-DbContext tạo các lớp thực thể cho mỗi bảng trong cơ sở dữ liệu SchoolDB và lớp Context với các cấu hình Fluent API cho tất cả các thực thể trong thư mục Models.



Sau đây là lớp thực thể Student được tạo cho bảng Student.

using System;

using System.Collections.Generic;

namespace EFCoreTutorials.Models

{

    public partial class Student

    {

        public Student()

        {

            StudentCourse = new HashSet<StudentCourse>();

        }

        public int StudentId { get; set; }

        public string FirstName { get; set; }

        public string LastName { get; set; }

        public int? StandardId { get; set; }

        public Standard Standard { get; set; }

        public StudentAddress StudentAddress { get; set; }

        public ICollection<StudentCourse> StudentCourse { get; set; }

    }

}

Sau đây là lớp SchoolDBContext mà bạn có thể sử dụng để lưu hoặc truy xuất dữ liệu.

using System;

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata;

namespace EFCoreTutorials.Models

{

    public partial class SchoolDBContext : DbContext

    {

        public virtual DbSet<Course> Course { get; set; }

        public virtual DbSet<Standard> Standard { get; set; }

        public virtual DbSet<Student> Student { get; set; }

        public virtual DbSet<StudentAddress> StudentAddress { get; set; }

        public virtual DbSet<StudentCourse> StudentCourse { get; set; }

        public virtual DbSet<Teacher> Teacher { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

        {

            if (!optionsBuilder.IsConfigured)

            {

#warning To protect potentially sensitive information in your connection string, you should move it out of source code. See http://go.microsoft.com/fwlink/?LinkId=723263 for guidance on storing connection strings.

                optionsBuilder.UseSqlServer(@"Server=.\SQLExpress;Database=SchoolDB;Trusted\_Connection=True;");

            }

        }

        protected override void OnModelCreating(ModelBuilder modelBuilder)

        {

            modelBuilder.Entity<Course>(entity =>

            {

                entity.Property(e => e.CourseName)

                    .HasMaxLength(50)

                    .IsUnicode(false);

                entity.HasOne(d => d.Teacher)

                    .WithMany(p => p.Course)

                    .HasForeignKey(d => d.TeacherId)

                    .OnDelete(DeleteBehavior.Cascade)

                    .HasConstraintName("FK\_Course\_Teacher");

            });

            modelBuilder.Entity<Standard>(entity =>

            {

                entity.Property(e => e.Description)

                    .HasMaxLength(50)

                    .IsUnicode(false);

                entity.Property(e => e.StandardName)

                    .HasMaxLength(50)

                    .IsUnicode(false);

            });

            modelBuilder.Entity<Student>(entity =>

            {

                entity.Property(e => e.StudentId).HasColumnName("StudentID");

                entity.Property(e => e.FirstName)

                    .HasMaxLength(50)

                    .IsUnicode(false);

                entity.Property(e => e.LastName)

                    .HasMaxLength(50)

                    .IsUnicode(false);

                entity.HasOne(d => d.Standard)

                    .WithMany(p => p.Student)

                    .HasForeignKey(d => d.StandardId)

                    .OnDelete(DeleteBehavior.Cascade)

                    .HasConstraintName("FK\_Student\_Standard");

            });

            modelBuilder.Entity<StudentAddress>(entity =>

            {

                entity.HasKey(e => e.StudentId);

                entity.Property(e => e.StudentId)

                    .HasColumnName("StudentID")

                    .ValueGeneratedNever();

                entity.Property(e => e.Address1)

                    .IsRequired()

                    .HasMaxLength(50)

                    .IsUnicode(false);

                entity.Property(e => e.Address2)

                    .HasMaxLength(50)

                    .IsUnicode(false);

                entity.Property(e => e.City)

                    .IsRequired()

                    .HasMaxLength(50)

                    .IsUnicode(false);

                entity.Property(e => e.State)

                    .IsRequired()

                    .HasMaxLength(50)

                    .IsUnicode(false);

                entity.HasOne(d => d.Student)

                    .WithOne(p => p.StudentAddress)

                    .HasForeignKey<StudentAddress>(d => d.StudentId)

                    .HasConstraintName("FK\_StudentAddress\_Student");

            });

            modelBuilder.Entity<StudentCourse>(entity =>

            {

                entity.HasKey(e => new { e.StudentId, e.CourseId });

                entity.HasOne(d => d.Course)

                    .WithMany(p => p.StudentCourse)

                    .HasForeignKey(d => d.CourseId)

                    .OnDelete(DeleteBehavior.ClientSetNull)

                    .HasConstraintName("FK\_StudentCourse\_Course");

                entity.HasOne(d => d.Student)

                    .WithMany(p => p.StudentCourse)

                    .HasForeignKey(d => d.StudentId)

                    .HasConstraintName("FK\_StudentCourse\_Student");

            });

            modelBuilder.Entity<Teacher>(entity =>

            {

                entity.Property(e => e.StandardId).HasDefaultValueSql("((0))");

                entity.Property(e => e.TeacherName)

                    .HasMaxLength(50)

                    .IsUnicode(false);

                entity.HasOne(d => d.Standard)

                    .WithMany(p => p.Teacher)

                    .HasForeignKey(d => d.StandardId)

                    .OnDelete(DeleteBehavior.Cascade)

                    .HasConstraintName("FK\_Teacher\_Standard");

            });

        }

    }

}

**Lưu ý:** EF Core chỉ tạo các lớp thực thể cho các bảng chứ không tạo cho Stored Procedure hoặc View.

##### ***DotNet CLI***

Nếu bạn sử dụng giao diện dòng lệnh dotnet để thực thi các lệnh EF Core thì hãy mở dấu nhắc lệnh và điều hướng đến thư mục gốc của dự án và thực hiện lệnh dotnet ef dbcontext scaffold sau:

> dotnet ef dbcontext scaffold "Server=.\SQLEXPRESS;Database=SchoolDB;Trusted\_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -o Models

Như vậy là bạn có thể tạo mô hình EF Core cho cơ sở dữ liệu có sẵn.

**Lưu ý:** Khi bạn đã tạo mô hình, bạn phải sử dụng các lệnh chuyển đổi (migration) bất cứ lúc nào bạn thay đổi mô hình để giữ cho cơ sở dữ liệu được cập nhật với mô hình.

## **DbContext**

Lớp DbContext là một phần không thể thiếu của Entity Framework. Một thể hiện của DbContext đại diện cho một phiên làm việc với cơ sở dữ liệu có thể được sử dụng để truy vấn và lưu các thể hiện của các thực thể của bạn vào cơ sở dữ liệu.

DbContext là sự kết hợp của các mẫu Unit Of Work và Repository. DbContext trong EF Core cho phép chúng ta thực hiện các tác vụ sau:

1. Quản lý kết nối cơ sở dữ liệu.
2. Cấu hình mô hình và mối quan hệ.
3. Truy vấn cơ sở dữ liệu.
4. Lưu dữ liệu vào cơ sở dữ liệu.
5. Cấu hình theo dõi thay đổi.
6. Bộ nhớ đệm.
7. Quản lý giao dịch.

Để sử dụng DbContext trong ứng dụng, chúng ta cần tạo lớp kế thừa từ lớp DbContext, còn được gọi là lớp Context.

Lớp Context này thường bao gồm các thuộc tính [Dbset <TEntity>](https://docs.microsoft.com/en-us/dotnet/api/microsoft.entityframeworkcore.dbset-1?view=efcore-2.0) cho mỗi thực thể trong mô hình. Hãy xem ví dụ sau mình họa về lớp Context trong EF Core.

public class SchoolContext : DbContext

{

    public SchoolContext()

    {

    }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

    {

    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)

    {

    }

    //entities

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

    public DbSet<Course> Courses { get; set; }

}

Trong ví dụ trên, lớp SchoolContext có nguồn gốc từ lớp DbContext và chứa các thuộc tính DbSet<TEntity> của các lớp thực thể Student và Course. Nó cũng ghi đè hai phương thức OnConfiguring và OnModelCreating.

Chúng ta phải tạo một thể hiện của SchoolContext để kết nối với cơ sở dữ liệu và lưu hoặc truy xuất dữ liệu Student hoặc Course.

Phương thức OnConfiguring() cho phép chúng ta lựa chọn và cấu hình nguồn dữ liệu được sử dụng với một Context sử dụng DbContextOptionsBuilder. Tìm hiểu cách cấu hình một lớp DbContext tại [đây](https://docs.microsoft.com/en-us/ef/core/miscellaneous/configuring-dbcontext) .

Phương thức OnModelCreating() cho phép chúng ta cấu hình mô hình sử dụng ModelBuilder Fluent API.

### ***Các phương thức của DbContext***

| **Phương thức** | **Mô tả** |
| --- | --- |
| Add | Thêm một thực thể mới vào DbContext với trạng thái Added và bắt đầu theo dõi nó. Dữ liệu thực thể mới này sẽ được chèn vào cơ sở dữ liệu khi gọi SaveChanges(). |
| AddAsync | Phương thức không đồng bộ để thêm một thực thể mới vào DbContext với trạng thái Added và bắt đầu theo dõi nó. Dữ liệu thực thể mới này sẽ được chèn vào cơ sở dữ liệu khi gọi SaveChangesAsync(). |
| AddRange | Thêm một tập các thực thể mới vào DbContext với trạng thái Added và bắt đầu theo dõi nó. Dữ liệu các thực thể mới này sẽ được chèn vào cơ sở dữ liệu khi gọi SaveChanges(). |
| AddRangeAsync | Phương thức không đồng bộ để thêm tập các thực thể mới vào DbContext với trạng thái Added và bắt đầu theo dõi nó. Dữ liệu thực thể mới này sẽ được chèn vào cơ sở dữ liệu khi gọi SaveChangesAsync(). |
| Attach | Đính kèm một thực thể mới hoặc đã tồn tại vào DbContext với trạng thái Unchanged và bắt đầu theo dõi nó. |
| AttachRange | Đính kèm một tập các thực thể mới hoặc đã tồn tại vào DbContext với trạng thái Unchanged và bắt đầu theo dõi nó. |
| Entry | Trả về một EntityEntry cho thực thể. Entry cung cấp quyền truy cập để thay đổi thông tin theo dõi và hoạt động cho thực thể. |
| Find | Tìm một thực thể với các giá trị khóa chính đã cho. |
| FindAsync | Phương thức không đồng bộ để tìm một thực thể với các giá trị khóa chính đã cho. |
| Remove | Thiết lập trạng thái Deleted cho thực thể được chỉ định sẽ xóa dữ liệu khi gọi SaveChanges(). |
| RemoveRange | Thiết lập trạng thái Deleted cho một tập các thực thể sẽ xóa dữ liệu trong một lần gọi DB duy nhất khi gọi SaveChanges(). |
| SaveChanges | Thực thi lệnh INSERT, UPDATE hoặc DELETE vào cơ sở dữ liệu cho các thực thể với trạng thái Added, Modified hoặc Deleted. |
| SaveChangesAsync | Phương thức SaveChanges() không đồng bộ |
| Set | Tạo một DbSet<TEntity> được sử dụng để truy vấn và lưu các thể hiện của TEntity. |
| Update | Đính kèm thực thể bị ngắt kết nối với trạng thái Modified và bắt đầu theo dõi nó. Dữ liệu sẽ được lưu khi gọi SaveChagnes(). |
| UpdateRange | Đính kèm một tập các thực thể bị ngắt kết nối với trạng thái Modified và bắt đầu theo dõi nó. Dữ liệu sẽ được lưu khi gọi SaveChagnes(). |
| OnConfiguring | Ghi đè phương thức này để cấu hình cơ sở dữ liệu (và các tùy chọn khác) sẽ được sử dụng cho Context này. Phương thức này được gọi khi mỗi thể hiện của Context được tạo. |
| OnModelCreating | Ghi đè phương thức này để cấu hình cho các thực thể trong các thuộc tính DbSet<TEntity> của lớp Context. |

### ***Các thuộc tính của DbContext***

| **Thuộc tính** | **Mô tả** |
| --- | --- |
| ChangeTracker | Cung cấp truy cập vào thông tin và hoạt động cho các thể hiện của thực thể mà Context này đang theo dõi. |
| Database | Cung cấp truy cập vào thông tin và hoạt động liên quan đến cơ sở dữ liệu cho Context này. |
| Model | Trả về siêu dữ liệu của các thực thể, mối quan hệ giữa chúng và cách chúng ánh xạ tới cơ sở dữ liệu. |

## **CRUD**

### ***Create (Add, AddRange, AddAsync, AddRangeAsync)***

Các đối tượng DbContext hay DbSet (như thuộc tính products của lớp ProductContext ở trên) có phương thức AddAsync để bạn chèn đối tượng phù hợp vào DbContext, nó nhận tham số là đối tượng Model cần chèn vào. Sau đó gọi phương thức SaveChangesAsync để thực hiện cập nhật dữ liệu vào Server SQL

// Thực hiện chèn hai dòng dữ liệu vào bảng Product

// Dùng AddAsync trong DbSet và trong DbContext

public static async Task InsertProduct()

{

    using (var context = new ProductsContext())

    {

        // Thêm sản phẩm 1

        await context.products.AddAsync(new Product

        {

            Name = "Sản phẩm 1",

            Provider = "Công ty 1"

        });

        // Thêm sản phẩm 2

        await context.AddAsync(new Product()

        {

            Name = "Sản phẩm 2",

            Provider = "Công ty 1"

        });

        // Thực hiện cập nhật thay đổi trong DbContext lên Server

        int rows = await context.SaveChangesAsync();

        Console.WriteLine($"Đã lưu {rows} sản phẩm");

    }

}

### ***Read***

Các đối tượng DbSet có phương thức ToListAsync() hay ToArrayAsync() để lấy về tất cả các dữ liệu (List) của bảng. Đặc biệt bạn có thể dùng LINQ ( đọc [Linq C#](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/)) trên các đối tượng này (nguồn là các DbSet)

public static async Task ReadProducts()

{

    using (var context = new ProductsContext())

    {

        // context.SetLogging();

        // Lấy danh sách các sản phẩm trong bảng

        var products = await context.products.ToListAsync();

        Console.WriteLine("Tất cả sản phẩm");

        foreach (var product in products)

        {

            Console.WriteLine($"{product.ProductId,2} {product.Name,10} - {product.Provider}");

        }

        Console.WriteLine();

        Console.WriteLine();

        // Dùng LINQ để truy vấn đến DbSet products (bảng product)

        // Lấy các sản phẩm cung cấp bởi CTY A

        products = await (from p in context.products

                          where (p.Provider == "CTY A")

                          select p

                         )

                        .ToListAsync();

        // Nếu không dùng bất đồng bộ chỗ này, có thể dùng

        // var pros = from p in context.products where (p.Provider == "CTY A") select p;

        Console.WriteLine("Sản phẩm CTY A");

        foreach (var product in products)

        {

            Console.WriteLine($"{product.ProductId,2} {product.Name,10} - {product.Provider}");

        }

    }

}

### ***Update***

Muốn cập nhật dữ liệu, chỉ việc thay đổi thuộc tính của đối tượng đọc được, sau đó gọi context.SaveChangesAsync()

// Đổi tên sản phẩm có ProductID thành  tên mới

public static async Task RenameProduct(int id, string newName)

{

    using (var context = new ProductsContext())

    {

        // Lấy  Product có  ID sản phẩm  chỉ  ra

        var product = await (from p in context.products where (p.ProductId == id) select p).FirstOrDefaultAsync();

        // Đổi tên và cập nhật

        if (product != null)

        {

            product.Name = newName;

            Console.WriteLine($"{product.ProductId,2} có tên mới = {product.Name,10}");

            await context.SaveChangesAsync();  //Thi hành cập nhật

        }

    }

}

Nếu muốn một đối tượng riêng lẻ, không giám sát - muốn thực hiện cập nhật thì gọi đến phương thức Update của DbSet

context.products.Update(p);

Nếu muốn cập nhật một số trường nào đó, dùng cách cập nhật đối tượng độc lập

// Tạo đối tượng

var pr = new Product()

{

    ProductId = 4,

    Name = "Abc"

};

// Gắn pr vào context để theo dõi, nó trả vể đối tượng EntityEntry<Product>

EntityEntry<Product> pr\_e = context.Attach(pr);

// Lấy thuộc tính Name của Product và thiết lập nó cần cập nhật

// với IsModified  = true;

pr\_e.Property(p => p.Name).IsModified = true;

context.SaveChanges();

Lưu ý, các đối tượng như Product kết quả truy vấn, hoặc được thêm vào ... thì mặc định là được giám sát - theo dõi bởi EF. Khi những đối tượng này thay đổi trạng thái thì context.SaveChanges() sẽ thực hiện các tác vụ dựa theo trạng thái của nó.

Mỗi dòng Product trả về từ truy vấn, hoặc thêm vào thì trong EF có một đối tượng EntityEntry tương ứng để để quản lý. Để lấy EntityEntry có thể dùng phương thức Entry, bạn có thể dùng nó để thay đổi trạng thái thủ công, ví dụ để điều chỉnh sự ảnh hưởng khi gọi SaveChange. Ví dụ, loại bỏ không bị giảm sát bởi EF cho đối tượng product nhận được

EntityEntry<Product> eProduct = context.Entry(product);

eProduct.State = EntityState.Detached;

Có một số state gồm: Added, Deleted, Detached, Modified, Unchanged

### ***Delete***

Để xóa dữ liệu khỏi DB, chỉ việc yêu cầu xóa đối tượng khỏi DbContext bằng phương thức Remove, rồi gọi SaveChangesAsync để cập nhật

// Xóa sản phẩm có ProductID = id

public static async Task DeleteProduct(int id)

{

    using (var context = new ProductsContext())

    {

        // context.SetLogging();

        var product = await (from p in context.products where (p.ProductId == id) select p).FirstOrDefaultAsync();

        if (product != null)

        {

            context.Remove(product);

            Console.WriteLine($"Xóa {product.ProductId}");

            await context.SaveChangesAsync();

        }

    }

}

## **EF Core Entity Configuration**

Đôi khi chúng ta muốn tùy chỉnh ánh xạ thực thể thành bảng cơ sở dữ liệu và không muốn tuân theo các quy ước mặc định.

EF Core cho phép chúng ta cấu hình các Entity class để tùy chỉnh ánh EF Model thành database. Mẫu lập trình này được gọi là [Convention over Configuration](https://en.wikipedia.org/wiki/Convention_over_configuration).

Có hai cách để cấu hình các lớp thực thể trong EF Core:

1. Cấu hình bằng cách sử dụng các Data Annotation Attribute.
2. Cấu hình bằng cách sử dụng Fluent API.

### ***Attribute chú thích dữ liệu trong EF Core***

Chú thích dữ liệu là một phương thức cấu hình dựa trên các attribute trong đó các attribute .NET khác nhau có thể được áp dụng cho các Entity class và các property để cấu hình mô hình.

Các Data Annotation Attribute không chỉ dành riêng cho Entity Framework, mà chúng cũng được sử dụng trong ASP.NET MVC.

Đây là lý do tại sao các attribute này được đặt trong namespace riêng biệt là [System.ComponentModel.DataAnnotations](https://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations(v=vs.110).aspx).

Ví dụ sau đây cho thấy cách các Data Annotation Attribute có thể được áp dụng cho một Entity class và các property để ghi đè các quy ước mặc định.

[Table("StudentInfo")]

public class Student

{

    public Student() { }

    [Key]

    public int SID { get; set; }

    [Column("Name", TypeName = "ntext")]

    [MaxLength(20)]

    public string StudentName { get; set; }

    [NotMapped]

    public int? Age { get; set; }

    public int StdId { get; set; }

    [ForeignKey("StdId")]

    public virtual Standard Standard { get; set; }

}

### ***Fluent API trong EF Core***

Một cách khác để cấu hình các Entity class là bằng cách sử dụng Entity Framework Fluent API. EF Fluent API dựa trên mẫu thiết kế Fluent API (hay còn gọi là [Fluent Interface](https://en.wikipedia.org/wiki/Fluent_interface)) trong đó kết quả được tạo thành từ một [method chaining](https://en.wikipedia.org/wiki/Method_chaining).

Trong Entity Framework Core, class [ModelBuilder](https://docs.microsoft.com/en-us/ef/core/api/microsoft.entityframeworkcore.modelbuilder) hoạt động như một Fluent API. Bằng cách sử dụng nó, chúng ta có thể cấu hình nhiều thứ khác nhau, vì nó cung cấp nhiều tùy chọn cấu hình hơn các Data Annotation Attribute.

Entity Framework Core Fluent API cấu hình các khía cạnh sau của một mô hình:

1. Cấu hình mô hình: Cấu hình EF Model để ánh xạ vào database. Cấu hình schema mặc định, các hàm DB, các data annotation attribute và thực thể bổ sung được loại trừ khỏi ánh xạ.
2. Cấu hình thực thể: Cấu hình Entity cho bảng và relationshops mapping, ví dụ: khóa chính, khóa thay thể (AlternateKey), Index, tên bảng, các mối quan hệ như một-một, một-nhiều, nhiều-nhiều, v.v.
3. Cấu hình thuộc tính: Cấu hình property để ánh xạ cột, ví dụ tên cột, giá trị mặc định, nullable, khóa ngoại, kiểu dữ liệu, cột chống xung đột, v.v.

Bảng sau liệt kê các phương thức quan trọng cho từng loại cấu hình.

| **Phương thức** | **Mô tả** |
| --- | --- |
| **Cấu hình mô hình** | |
| HasDbFunction | Cấu hình chức năng cơ sở dữ liệu. |
| HasDefaultSchema | Chỉ định lược đồ cơ sở dữ liệu mặc định. |
| HasAnnotation | Thêm hoặc cập nhật các attribute chú thích dữ liệu trên thực thể. |
| HasSequence | Cấu hình chuỗi cơ sở dữ liệu. |
| **Cấu hình thực thể** | |
| HasAlternateKey | Cấu hình khóa thay thế trong mô hình EF cho thực thể. |
| HasIndex | Cấu hình thuộc tính Index cho kiểu thực thể. |
| HasKey | Cấu hình thuộc tính khóa chính cho kiểu thực thể. |
| HasMany | Cấu hình phần nhiều của mối quan hệ một-nhiều hoặc nhiều-nhiều. |
| HasOne | Cấu hình phần một của mối quan hệ một-nhiều hoặc một-một. |
| Ignore | Cấu hình lớp thực thể hoặc thuộc tính không được ánh xạ vào bảng hoặc cột trong cơ sở dữ liệu. |
| OwnsOne | Cấu hình mối quan hệ trong đó thực thể đích được sở hữu bởi thực thể này. Giá trị khóa thực thể đích được truyền từ thực thể mà nó thuộc về. |
| ToTable | Cấu hình tên bảng cho thực thể. |
| **Cấu hình thuộc tính** | |
| HasColumnName | Cấu hình tên cột tương ứng của một thuộc tính trong cơ sở dữ liệu. |
| HasColumnType | Cấu hình kiểu dữ liệu của cột tương ứng của một thuộc tính trong cơ sở dữ liệu. |
| HasComputedColumnSql | Cấu hình thuộc tính để ánh xạ tới cột được tính toán trong cơ sở dữ liệu. |
| HasDefaultValue | Cấu hình giá trị mặc định cho cột mà thuộc tính ánh xạ. |
| HasDefaultValueSql | Cấu hình biểu thức giá trị mặc định cho cột mà thuộc tính ánh xạ. |
| HasField | Chỉ định trường sao lưu được sử dụng với thuộc tính. |
| HasMaxLength | Cấu hình độ dài tối đa của dữ liệu có thể được lưu trữ trong một thuộc tính. |
| IsConcurrencyToken | Cấu hình thuộc tính được sử dụng làm mã thông báo xung đột. |
| IsRequired | Cấu hình thuộc tính bắt buộc nhập dữ liệu khi gọi phương thức SaveChanges. |
| IsRowVersion | Cấu hình thuộc tính được sử dụng để phát hiện xung đột. |
| IsUnicode | Cấu hình thuộc tính chuỗi có thể chứa các ký tự unicode hoặc không. |
| ValueGeneratedNever | Cấu hình một thuộc tính không thể có giá trị được tạo tự động khi thực thể được lưu. |
| ValueGeneratedOnAdd | Cấu hình rằng thuộc tính có giá trị được tạo tự động khi lưu thực thể mới. |
| ValueGeneratedOnAddOrUpdate | Cấu hình rằng thuộc tính có giá trị được tạo tự động khi lưu thực thể mới hoặc cập nhật thực thể hiện có. |
| ValueGeneratedOnUpdate | Cấu hình rằng thuộc tính có giá trị được tạo tự động khi cập nhật thực thể hiện có. |

### ***Cấu hình Fluent API trong EF Core***

Ghi đè phương thức OnModelCreating và sử dụng tham số modelBuilder kiểu ModelBuilder để cấu hình các Entity class như ví dụ bên dưới.

public class SchoolDBContext : DbContext

{

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)

    {

        //Write Fluent API configurations here

        //Property Configurations

        modelBuilder.Entity<Student>()

                .Property(s => s.StudentId)

                .HasColumnName("Id")

                .HasDefaultValue(0)

                .IsRequired();

    }

}

Trong ví dụ trên, thể hiện của ModelBuilder được sử dụng để cấu hình một thuộc tính bằng cách gọi một chuỗi nhiều phương thức.

Nó cấu hình property StudentId của thực thể Student:

* StudentId có tên là Id trong cơ sở dữ liệu bằng cách sử dụng phương thức HasColumnName
* StudentId có giá trị mặc định là 0 bằng cách sử dụng phương thức HasDefaultValue
* StudentId cho phép nullable bằng phương thức IsRequired

Các cấu hình trên được thực hiện trong một câu lệnh thay vì nhiều câu lệnh. Điều này làm tăng khả năng đọc và cũng mất ít thời gian hơn để viết so với nhiều câu lệnh, như được trình bày bên dưới.

//Fluent API method chained calls

modelBuilder.Entity<Student>()

        .Property(s => s.StudentId)

        .HasColumnName("Id")

        .HasDefaultValue(0)

        .IsRequired();

//Separate method calls

modelBuilder.Entity<Student>().Property(s => s.StudentId).HasColumnName("Id");

modelBuilder.Entity<Student>().Property(s => s.StudentId).HasDefaultValue(0);

modelBuilder.Entity<Student>().Property(s => s.StudentId).IsRequired();

**Lưu ý:** Cấu hình Fluent API có quyền ưu tiên cao hơn các Data Annotation Attribute.

### ***Truy vấn trong Entity Framework Core***

Truy vấn trong Entity Framework Core vẫn giống như [truy vấn trong Entity Framework 6.x](https://comdy.vn/entity-framework/truy-van-trong-entity-framework/), với các truy vấn SQL được tối ưu hóa hơn và khả năng đưa các phương thức C# / VB.NET vào các truy vấn LINQ-to-Entities.

Ở phần này, bạn sẽ tìm hiểu các tính năng mới của truy vấn được giới thiệu trong Entity Framework Core.

#### **Phương thức C# / VB.NET trong truy vấn**

EF Core có một tính năng mới trong LINQ-to-Entities nơi chúng ta có thể thêm các phương thức C# hoặc VB.NET trong truy vấn. Điều này là không thể trong EF 6.

private static void Main(string[] args)

{

    var context = new SchoolContext();

    var studentsWithSameName = context.Students

                                      .Where(s => s.FirstName == GetName())

                                      .ToList();

}

public static string GetName()

{

    return "Bill";

}

#### **Eager Loading trong EF Core**

**Eager Loading là gì?**

Eager Loading là quá trình trong đó một truy vấn cho một Entity cũng load các Entity liên quan như một phần của truy vấn, do đó chúng ta không cần phải thực hiện một truy vấn riêng cho các Entity liên quan.

Entity Framework Core hỗ trợ Eager Loading các Entity liên quan, bằng cách sử dụng phương thức mở rộng Include() và truy vấn tạo kết quả đầu ra.

Ngoài ra, nó cũng cung cấp phương thức mở rộng ThenInclude() để tải nhiều cấp độ của các Entity liên quan.

**Phương thức Include trong EF Core**

Chúng ta có thể chỉ định biểu thức lambda làm tham số trong phương thức Include() để chỉ định thuộc tính điều hướng như dưới đây.

var context = new SchoolContext();

var studentWithGrade = context.Students

                           .Where(s => s.FirstName == "Bill")

                           .Include(s => s.Grade)

                           .FirstOrDefault();

**Phương thức ThenInclude trong EF Core**

EF Core đã giới thiệu phương thức mở rộng ThenInclude() mới để tải nhiều cấp độ của các Entity liên quan. Hãy xem xét ví dụ sau:

var context = new SchoolContext();

var student = context.Students.Where(s => s.FirstName == "Bill")

                        .Include(s => s.Grade)

                            .ThenInclude(g => g.Teachers)

                        .FirstOrDefault();

**Truy vấn tạo kết quả đầu ra**

Chúng tôi cũng có thể tải nhiều Entity liên quan bằng cách sử dụng truy vấn tạo kết quả đầu ra thay vì sử dụng phương thức Include() hoặc ThenInclude().

Ví dụ sau đây cho thấy các truy vấn tạo kết quả đầu ra để tải các Entity Student, Grade và Teacher.

var context = new SchoolContext();

var stud = context.Students.Where(s => s.FirstName == "Bill")

                        .Select(s => new

                        {

                            Student = s,

                            Grade = s.Grade,

                            GradeTeachers = s.Grade.Teachers

                        })

                        .FirstOrDefault();

#### **Lazy loading trong EF Core**

Lazy loading trì hoãn việc tải các dữ liệu liên quan, cho đến khi bạn yêu cầu cụ thể. Lazy loading đối lập hoàn toàn với Eager loading.

Lazy loading được giới thiệu trong EF Core 2.1 như một tính năng tùy chọn bổ sung.

Lazy loading có thể được kích hoạt theo hai cách:

* Sử dụng Proxies
* Sử dụng interface ILazyLoader

#### **Explicit Loading trong EF Core**

Ở phần này bạn sẽ tìm hiểu làm thế nào để tải các Entity liên quan trong một biểu đồ thực thể rõ ràng.

Sử dụng phương thức Load() để tải các Entity liên quan một cách rõ ràng. Hãy xem xét ví dụ sau.

using (var context = new SchoolContext())

{

    var student = context.Students

                         .Where(s => s.FirstName == "Bill")

                         .FirstOrDefault<Student>();

    // loads StudentAddress

    context.Entry(student).Reference(s => s.StudentAddress).Load();

    // loads Courses collection

    context.Entry(student).Collection(s => s.StudentCourses).Load();

}

## **DI EF Core**

Thực hiện thêm các package để làm việc với EF kết nối đến MS SQL Server và các công cụ trợ giúp phát sinh code:

dotnet tool install --global dotnet-ef

dotnet tool install --global dotnet-aspnet-codegenerator

dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design

dotnet add package Microsoft.EntityFrameworkCore.Design

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

sau đó viết chuỗi kết nối vào file appsettings.json để sau này EF sử dụng.

{

  "Logging": {

    "LogLevel": {

      "Default": "Information",

      "Microsoft.AspNetCore": "Warning"

    }

  },

  "AllowedHosts": "\*",

  "ConnectionStrings": {

    "MyBlogDb": " Data Source = localhost,1433; Initial Catalog = myblog; User ID = SA; Password = Password123

"

  }

}

Trong đó, MyBlogDb nên để là tên Database, trong Ví dụ này, Db name là MyBlog

Tiếp theo vào file Program.cs để đăng ký DbContext

using EFCore.Infras;

using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

var configuration = builder.Configuration;

var services = builder.Services;

// Add services to the container.

services.AddControllers();

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle

services.AddEndpointsApiExplorer();

services.AddSwaggerGen();

services.AddDbContext<MyBlogDbContext>(options =>

    options.UseSqlServer(configuration.GetConnectionString("MyBlogDb")));

var app = builder.Build();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

    app.UseSwagger();

    app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

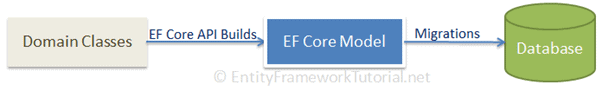
app.MapControllers();

app.Run();

## **Migration**

Migration là kỹ thuật trong việc tương tác với cơ sở dữ liệu, theo đó việc thay đổi về cấu trúc database ở code sẽ được cập nhật lên database đảm bảo dữ liệu đang tồn tại không bị mất, lịch sử (phiên bản) cập nhật được lưu lại sau mỗi lần cập nhật.

Thường khi sử dụng EF làm việc với DB, có hai cách đó là làm việc với một database đang tồn tại (gọi là database first) - việc cập nhật database thực hiện khá độc lập với ứng dụng - tình huống này Migration ít hữu ích, tuy nhiên trường hợp bạn tạo database từ code, thay đổi cấu trúc database ... bằng code thì migration rất hữu ích. Tất nhiên ta vẫn có cách để sử dụng EF Migration trên database đã tồn tại.

[](https://www.entityframeworktutorial.net/images/efcore/ef-core-migration.png)

Với migration khi bạn cập nhật Model, yêu cầu database cập nhật thì nó sẽ lưu thông tin phiên bản hiện tại của cấu trúc Model (database) ở Server DB - ví dụ phiên bản a, sau đó thay đổi các Model, lại yêu cầu cập nhật thì nó sẽ đọc thông tin phiên bản cuối trên DB, so sánh sự khác biệt và cập nhật sự khác biệt đó để lên phiên bản mới, phiên bản b.

EF Core Migration dùng các lệnh để thực thi trong gói  **NuGet Package Manager Console** hoặc có thể sử dụng **dotnet Command Line Interface (CLI).**

Dưới đây là danh sách các **Migration**quan trọng trong EF Core

| **Lệnh PMC** | **Lệnh  dotnet CLI** | **Cách sử dụng** |
| --- | --- | --- |
| add-migration <migration name> | Add <migration name> | Sẽ tạo ra một lớp thực hiện việc chuyển đổi dữ liệu tương thích với sự thay đổi model của bạn ngay bên trong dự án. . |
| Remove-migration | Remove | Xóa migration mới nhất. |
| Update-database | Update | sẽ cập nhật CSDL để CSDL có các bảng tương ứng với sự thay đổi của các Model |
| Script-migration | Script | Tạo SQL Script cho Migration. |

### ***Tạo một Migration***

Trước tiên, Chúng ta phải định nghĩa  các domain class (Entity class) ban đầu. Tại thời điểm này, không có database nào cho ứng dụng của chúng ta có thể lưu trữ dữ liệu từ các domain class. Vì vậy, trước tiên, chúng ta cần tạo một migration.

Mở **Package Manager Console** từ **Tools -> NuGet Package Manager -> Package Manager Console.**

**Package Manager Console**

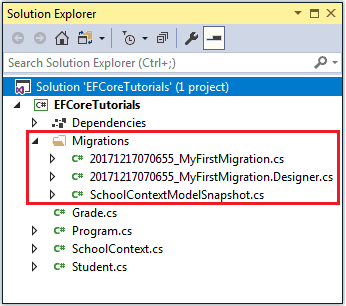
PM> add-migration MyFirstMigration

Nếu sử dụng lệnh dotnet**Command Line Interface**.

**CLI**

> dotnet ef migrations add MyFirstMigration

Sau lệnh này, nó tạo ra 3 file trong thư mục Migrations các file có tên dạng:

[](https://www.entityframeworktutorial.net/images/efcore/ef-core-migration2.png)

Số **20171217070655**sinh ra theo thời điểm chạy lệnh. Ba file này chứa thông tin để có thể cập nhật (hoặc tạo) database đúng cấu trúc Model ở thời điểm mà bạn tạo Migration.

1. **<timestamp>\_<Migration Name>.cs:**một Migration có một lớp kế thừa từ lớp Migration được tạo ra, trong nó có hai phương thức là Up và Down - để thực hiện chuyển từ phiên bản thấp đến phiên bản này (Up) hoặc đang từ phiên bản này lùi về phiên bản trước (Down). Lớp này được định nghĩa trong 2 file mã nguồn còn lại
2. **<timestamp>\_<Migration Name>.Designer.cs:** Tệp siêu dữ liệu của migration chứa thông tin được EF Core sử dụng.
3. **<contextclassname>ModelSnapshot.cs:**  Là một ảnh chụp model hiện hành. Được sử dụng để quyết định những gì đã thay đổi khi tạo lần migration tiếp theo.

Bây giờ, sau khi tạo migration snapshot, đã đến lúc tạo database.

### ***Tạo hoặc Cập nhật cơ sở dữ liệu***

Sử dụng lệnh sau để tạo hoặc cập nhật database schema.

**Package Manager Console**

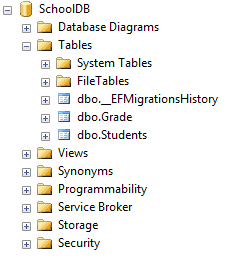
PM> Update-Database

**CLI**

> dotnet ef database update

Lệnh Cập nhật sẽ tạo database dựa trên context và các domain class (Entity class) và migration snapshot, được tạo bằng cách sử dụng lệnh **add-migration**.

Nếu đây là lần migration đầu tiên, thì nó cũng sẽ tạo ra một bảng có tên **\_\_EFMigrationsHistory**, bảng này sẽ lưu trữ tên của tất cả các lần migration cũng như thời điểm chúng sẽ được áp dụng cho database.

[](https://www.entityframeworktutorial.net/images/efcore/ef-core-migration3.png)

### ***Xóa một Migration***

Chúng ta có thể xóa migration mới nhất nếu không được thêm vào database. Sử dụng lệnh xóa sau:

**Package Manager Console**

PM> remove-migration

**CLI**

> dotnet ef migrations remove

Các lệnh trên sẽ xóa migration mới nhất và trả về model snapshot cho migration kế tiếp. Chú ý: nếu migration tồn tại trong database nó sẽ ném ra lỗi ngoại lệ.

### ***Quay trở lại một Migration***

Giả sử bạn đã thay đổi domain class (Entity class) của mình và tạo **migration**thứ hai có tên **MySecondMigration**bằng cách sử dụng lệnh **add-migration**và áp dụng **migration**này cho cơ sở dữ liệu bằng lệnh **Update**. Tuy nhiên, vì lý do nào đó, bạn muốn quay trở lại trạng thái ban đầu cho database. Trong trường hợp này, hãy sử dụng lệnh **update-database <migration name>** để quay trở về database về **migration snapshot** trước đó.

**Package Manager Console**

PM> Update-database MyFirstMigration

**CLI**

> dotnet ef database update MyFirstMigration.

Lệnh trên quay trở lại database ban đầu dựa vào lệnh **migration**với tên **MyFirstMigration**và xóa tất cả thay đổi đã thực thi cho lần migration thứ 2 **MySecondMigration.**Đồng thời cũng xóa **MySecondMigration**từ bảng **\_\_EFMigrationsHistory**trong database.

Chú ý: Thao tác này sẽ không xóa các tệp migration liên quan đến **MySecondMigration**. Sử dụng các lệnh **remove**để xóa chúng khỏi dự án.

### ***Tạo Script SQL***

Sử dụng lệnh sau để tạo Script SQL

**Package Manager Console**

PM> script-migration

**CLI**

> dotnet ef migrations script

Bạn có thể quay về một phiên bản bất kỳ trong danh sách bằng cách thực hiện lệnh dotnet ef database update tên\_quay\_về

## **Run Sql query with EF Core**

Entity Framework Core cung cấp phương thức **DbSet.FromSql ()**để thực thi các truy vấn SQL trong database bên dưới và nhận kết quả dưới dạng các entity object.

Ví dụ sau minh họa việc thực thi một truy vấn SQL trong database MS SQL Server

var context = new SchoolContext();

var students = context.Students

                  .FromSql("Select \* from Students where Name = 'Bill'")

                  .ToList();

Trong ví dụ trên, phương thức **FromSql()** được sử dụng sau collection entity Student (**DbSet <Student >**), vì vậy truy vấn SQL được chỉ định phải trả về các bản ghi từ bảng **Student**. Entity Framework Core sẽ thực hiện truy vấn được xác định đến database, tức là **Select \* from Students where Name = 'Bill'** trong ví dụ trên.

### ***Truy vấn có tham số***

Phương thức FromSql cho phép các truy vấn có tham số bằng cách sử dụng cú pháp nội suy chuỗi trong C #, như được hiển thị bên dưới.

string name = "Bill";

var context = new SchoolContext();

var students = context.Students

                    .FromSql($"Select \* from Students where Name = '{name}'")

                    .ToList();

**Sau đây cũng đúng.**

string name = "Bill";

var context = new SchoolContext();

var students = context.Students

                    .FromSql("Select \* from Students where Name = '{0}'", name)

                    .ToList();

Các ví dụ trên sẽ thực thi truy vấn SQL sau đối với cơ sở dữ liệu SQL Server:

exec sp\_executesql N'Select \* from Students where Name = ''@p0''

',N'@p0 nvarchar(4000)',@p0=N'Bill'

go

### ***Các toán tử LINQ***

Chúng ta cũng có thể sử dụng toán tử LINQ sau một truy vấn bằng phương pháp FromSql.

string name = "Bill";

var context = new SchoolContext();

var students = context.Students

                    .FromSql("Select \* from Students where Name = '{0}'", name)

                    .OrderBy(s => s.StudentId)

                    .ToList();

Trong ví dụ trên, EF Core thực hiện truy vấn sau bằng cách kết hợp phương thức FromSql và toán tử OrderBy.

exec sp\_executesql N'SELECT [s].[StudentId], [s].[Name]

FROM (

Select \* from Students where Name = ''@p0''

) AS [s]

ORDER BY [s].[StudentId]',N'@p0 nvarchar(4000)',@p0=N'Bill'

go

### ***Hạn chế của FromSql***

1. Các truy vấn SQL phải trả về các Entity cùng kiểu với kiểu **DbSet <T>**. ví dụ. truy vấn xác định không thể trả về các thực thể **Course**nếu **FromSql** được sử dụng sau **Student**. Trả về các kiểu đặc biệt từ phương thức **FromSql ()** chứa trong [backlog](https://github.com/aspnet/EntityFrameworkCore/issues/1862).
2. Truy vấn SQL phải trả về tất cả các cột của bảng. ví dụ. **context.Students.FromSql ("Chọn StudentId, LastName từ Students) .ToList ()** sẽ ném ra một ngoại lệ.
3. Truy vấn SQL không thể bao gồm các truy vấn JOIN để lấy dữ liệu liên quan. Sử dụng phương thức **Include**để tải các thực thể liên quan sau phương thức **FromSql ()**.

## **Exam**

### Bài 1:

Tạo database như sau:

* Dbname: EFCore\_Exam1
* Table Product:
  + Id: number – primary key – auto increament
  + Name: nvarchar
  + Price: number
  + Status: number
  + CategoryId: number
  + CreateDate: DateTime
* Table Category:
  + Id: number – primary key – auto increament
  + Name: nvarchar
  + Status: number
* Stored Procedure:
  + Sp\_getallproduct: param là categoryId, nếu truyền null thì getAll
  + Sp\_deleteProduct: param là Id, nếu truyền null thì delete all
  + Sp\_insertProduct

Tạo project web api sử dụng EFCore với kiểu Database first kết nối tới Database EFCore\_Exam1.

Gồm 2 controller: StoredProcedureController và CrudController.

* StoredProcedureController: Có các Api gọi đến các thủ tục của database
* CrudController: Có các Api Crud trực tiếp bằng EFCore

**Lưu ý: Phần controller không nên chứa code logic**

### Bài 2:

Tạo Web Api với yêu cầu sau:

* Model Student
  + Id: int
  + Name: string
  + Birthday: Datetime
  + Gender: int
  + Status: bool
* Model Subject
  + Id: int
  + Name: string
  + Status: bool
* Model Mark
  + StudentId: int
  + SubjectId: int
  + Scores: int
  + CreateDate: DateTime
* DbContext: StudentDbContext

Yêu cầu:

Dùng EFCore Code First tạo Database với config như sau:

* Model Student => table “tbl\_student”
  + Id => column “StudentId”: primary key, auto increament
  + Name => column “StudentName”: size 250, not null
  + Birthday => column “Birthday”: not null
  + Gender => column “Gender”: Not null, default 0
  + Status => column “Status”: Not null, default true
* Model Subject => table “tbl\_subject”
  + Id => column “SubjectId”: primary key, auto increament
  + Name => column “SubjectName”: not null, size 200
  + Status => column “Status”: Not null, default true
* Model Mark => table “tbl\_mark”
  + StudentId => column “StudentId”: foreign key, not null
  + SubjectId => column “SubjectId”: foreign key, not null
  + Scores => column “Scores”: not null, default 0
  + CreateDate => column “CreateDate”: not null
* Config relationship giữa các entity:
  + 1 Student 🡪 n Mark
  + 1 Subject 🡪 n Mark
* Config khi delete Student thì sẽ delete luôn dữ liệu liên quan đến student ở các table khác
* Database name “EFCore\_Exa2”

Web Api với các chức năng sau:

* Sử dụng Net
* Các Api Crud của các model
* Mỗi model 1 controller
* Logic không để ở controller
* Api khác:
  + Lấy data của Student bằng Id: hiển thị data của student và data Mark liên quan đến student
  + Lấy dữ liệu Subject bằng Id: có option cho phép hiển thị cả dữ liệu cả Mark liên quan đến Subject và dữ liệu Student liên quan đến Mark tương ứng.

### Bài 3:

các yêu cầu như bài 2 nhưng sử dụng Database Oracle

**Lưu ý: Code phải try/catch đầy đủ**