Entity Framework Core

Entity Framework Core

- Entity Framework (EF) Core is a lightweight, extensible, open source and cross-platform version of the popular Entity Framework data access technology.
- EF Core can serve as an object-relational mapper (O/RM), enabling .NET developers to work with a database using .NET objects, and eliminating the need for most of the data-access code they usually need to write.
- You can generate a model from an existing database, hand code a model to match your database, or use EF Migrations to create a database from your model, and then evolve it as your model changes over time.

Install Entity Framework Core

- Tools ▸ NuGet Package Manager ▸ Package Manager Console
- Run
```
Install-Package Microsoft.EntityFrameworkCore.SqlServer
```
- To enable reverse engineering from an existing database we need to install a couple of other packages:
- Run
```
Install-Package Microsoft.EntityFrameworkCore.Tools
```

Working With EF Core (Code First From DB)

- Tools –> NuGet Package Manager –> Package Manager Console.
- Run the following command to create a model from the existing database:

```
Scaffold-DbContext "Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;“
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

Using EF Core’s AppDbContext(Code First)

1- install Entity Framework Core NuGet packages.

2- Create the data model

3- Create the Database Context

4-Add your connection string to Database Context

5-Add Migration and Update Database.

Configure Database Context

- protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(@"Server=(localdb)\\mssqllocaldb; Database=EFGetStarted.ConsoleApp.NewDb;Trusted_Connection=True; TrustServerCertificate=True;”)

Loading Related Data

Entity Framework Core allows you to use the navigation properties in your model to load related entities. There are three common O/RM patterns used to load related data.

- **Eager loading** means that the related data is loaded from the database as part of the initial query.
- **Explicit loading** means that the related data is explicitly loaded from the database at a later time.
- **Lazy loading** means that the related data is transparently loaded from the database when the navigation property is accessed.

Eager loading

- You can use the ***Include*** method to specify related data to be included in query results.

```
var blogs = context.Blogs
    .Include(blog => blog.Posts)
    .ToList();
```

- You can include related data from multiple relationships in a single query.

```
var blogs = context.Blogs
    .Include(blog => blog.Posts)
    .Include(blog => blog.Owner)
    .ToList();
```

Including multiple levels -Eager loading

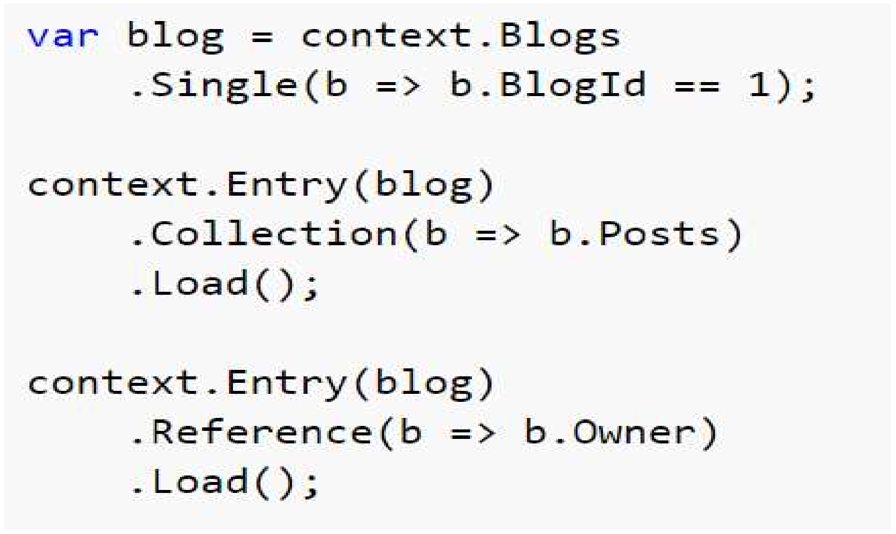
- You can drill down thru relationships to include multiple levels of related data using the *ThenInclude* method.

```
var blogs = context.Blogs
    .Include(blog => blog.Posts)
        .ThenInclude(post => post.Author)
    .ToList();
```

```
var blogs = context.Blogs
    .Include(blog => blog.Posts)
        .ThenInclude(post => post.Author)
            .ThenInclude(author => author.Photo)
    .Include(blog => blog.Owner)
        .ThenInclude(owner => owner.Photo)
    .ToList();
```

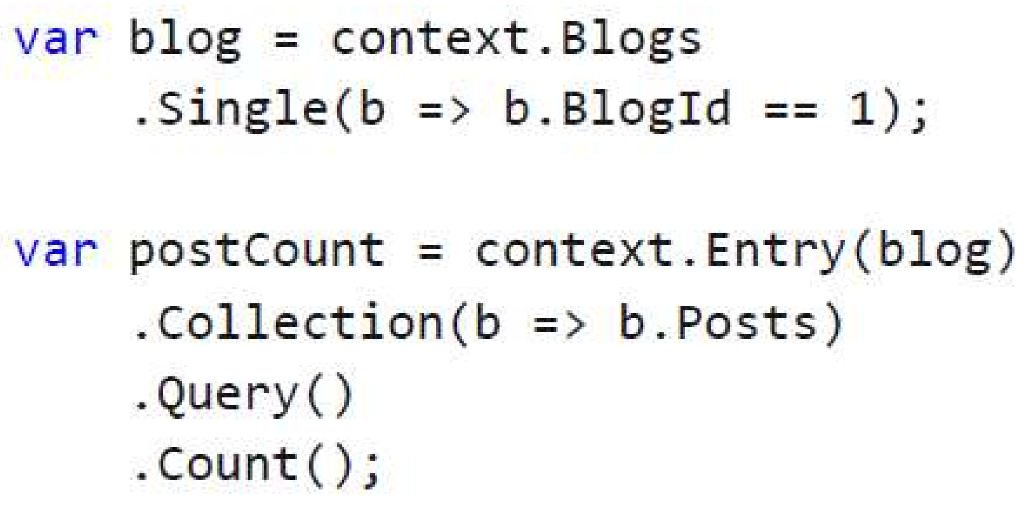
Explicit loading

- This feature was introduced in EF Core 1.1.
- When the entity is first read, related data isn't retrieved. You write code that retrieves the related data if it's needed.
- You can explicitly load a navigation property via the DbContext.Entry(...) API.

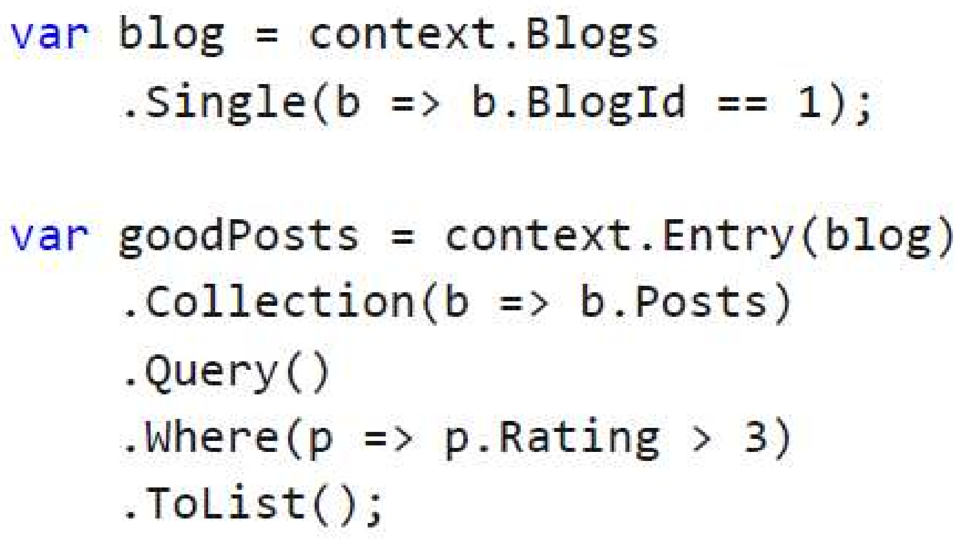


```
var blog = context.Blogs
    .Single(b => b.BlogId == 1);

context.Entry(blog)
    .Collection(b => b.Posts)
    .Load();

context.Entry(blog)
    .Reference(b => b.Owner)
    .Load();
```

Explicit loading



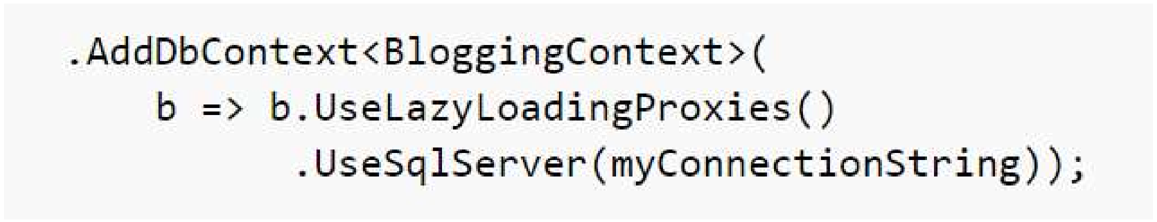
```
var blog = context.Blogs
    .Single(b => b.BlogId == 1);

var postCount = context.Entry(blog)
    .Collection(b => b.Posts)
    .Query()
    .Count();
```



```
var blog = context.Blogs
    .Single(b => b.BlogId == 1);

var goodPosts = context.Entry(blog)
    .Collection(b => b.Posts)
    .Query()
    .Where(p => p.Rating > 3)
    .ToList();
```

Lazy loading

- This feature was introduced in **EF Core 2.1**.
- The simplest way to use lazy-loading is by installing the Microsoft.EntityFrameworkCore.Proxies package and enabling it with a call to *UseLazyLoadingProxies* when using AddDbContext:



```
.AddDbContext<BloggingContext>(
    b => b.UseLazyLoadingProxies()
          .UseSqlServer(myConnectionString));
```

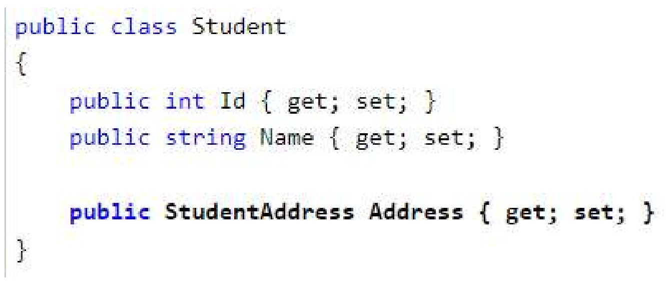
EF Core will then enable lazy-loading for any navigation property that can be overridden--that is, it must be *virtual* and on a class that can be inherited from.

Performance considerations

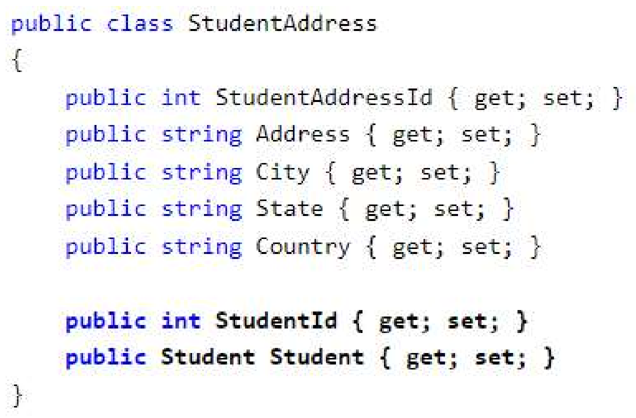
- If you know you need related data for every entity retrieved, eager loading often offers the best performance, because a single query sent to the database is typically more efficient than separate queries for each entity retrieved.
- In some scenarios separate queries is more efficient. Eager loading of all related data in one query might cause a very complex join to be generated, which SQL Server can't process efficiently.

One-To-One relation

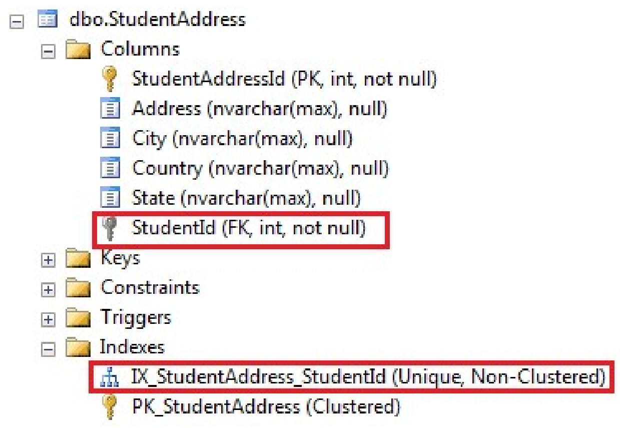
- Entity Framework Core introduced default conventions which automatically configure a One-to-One relationship between two entities



```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }

    public StudentAddress Address { get; set; }
}
```



```
public class StudentAddress
{
    public int StudentAddressId { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Country { get; set; }

    public int StudentId { get; set; }
    public Student Student { get; set; }
}
```

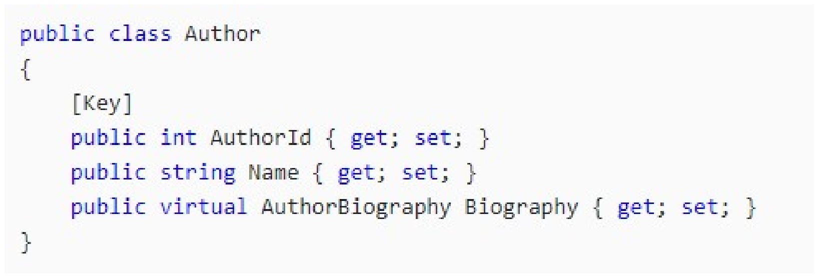


```
⊟ dbo.StudentAddress
  ⊟ Columns
       StudentAddressId (PK, int, not null)
       Address (nvarchar(max), null)
       City (nvarchar(max), null)
       Country (nvarchar(max), null)
       State (nvarchar(max), null)
       StudentId (FK, int, not null)
  ⊞ Keys
  ⊞ Constraints
  ⊞ Triggers
  ⊟ Indexes
       IX_StudentAddress_StudentId (Unique, Non-Clustered)
       PK_StudentAddress (Clustered)
```

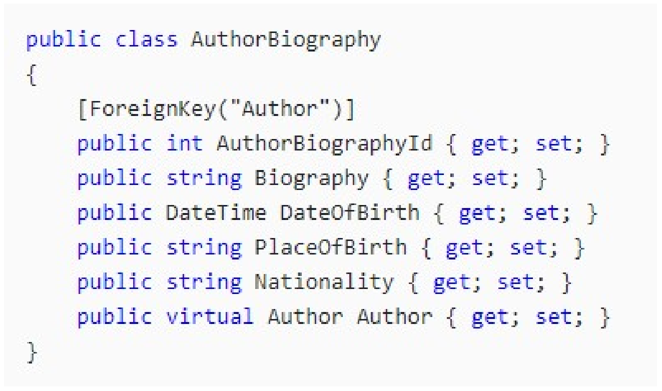
One-To-One relation EF6

- In a one-to-one relationship, each row of data in one table is linked to zero or one row in the second table

1-Using Data Annotation



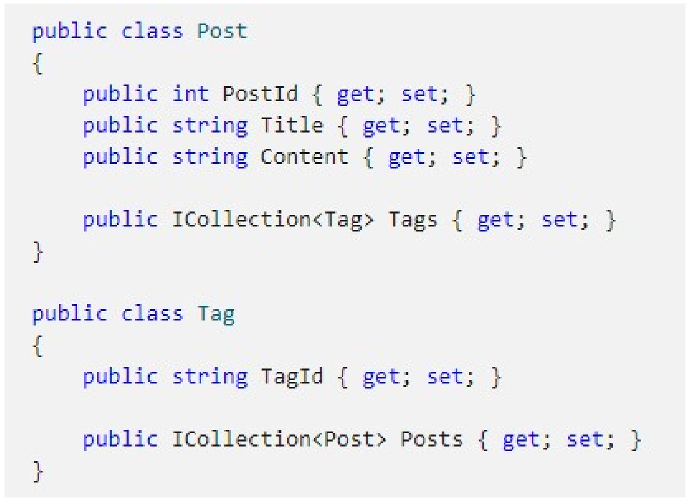
```
public class Author
{
    [Key]
    public int AuthorId { get; set; }
    public string Name { get; set; }
    public virtual AuthorBiography Biography { get; set; }
}
```



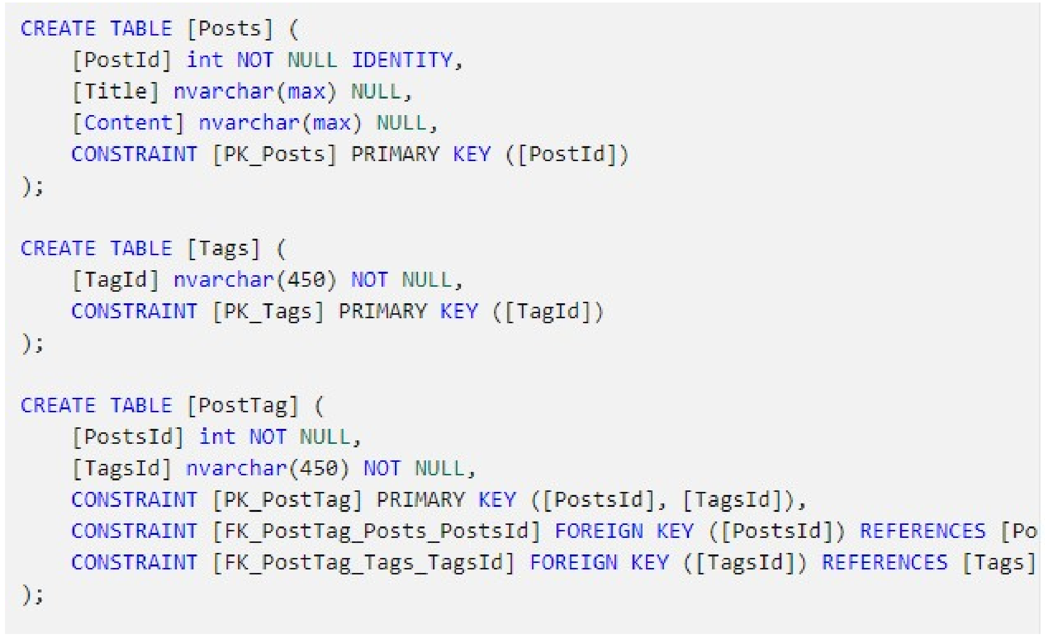
```
public class AuthorBiography
{
    [ForeignKey("Author")]
    public int AuthorBiographyId { get; set; }
    public string Biography { get; set; }
    public DateTime DateOfBirth { get; set; }
    public string PlaceOfBirth { get; set; }
    public string Nationality { get; set; }
    public virtual Author Author { get; set; }
}
```

M-to-M

Many to many relationships require a collection navigation property on both sides. They will be discovered by convention like other types of relationships.



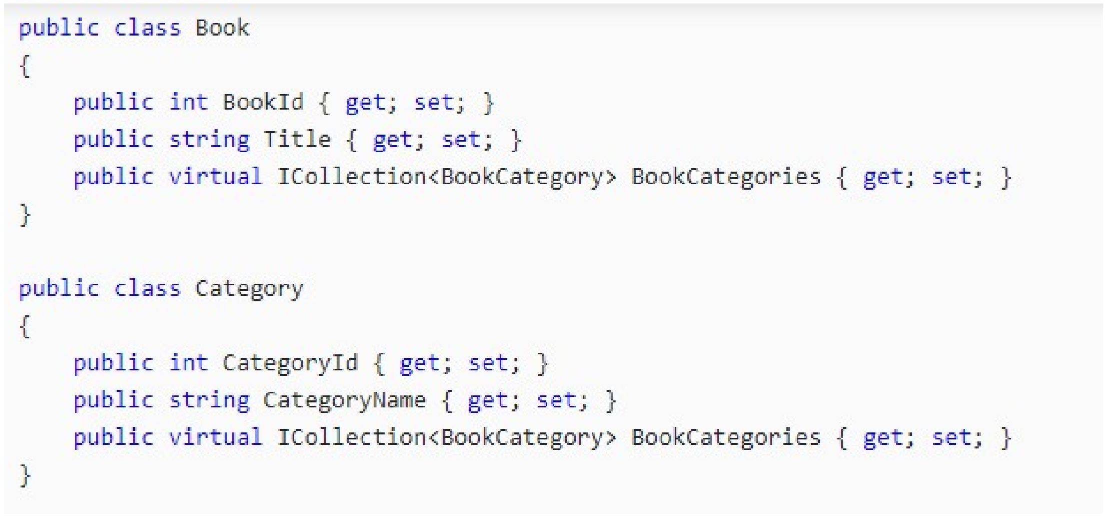
```
public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public ICollection<Tag> Tags { get; set; }
}

public class Tag
{
    public string TagId { get; set; }

    public ICollection<Post> Posts { get; set; }
}
```



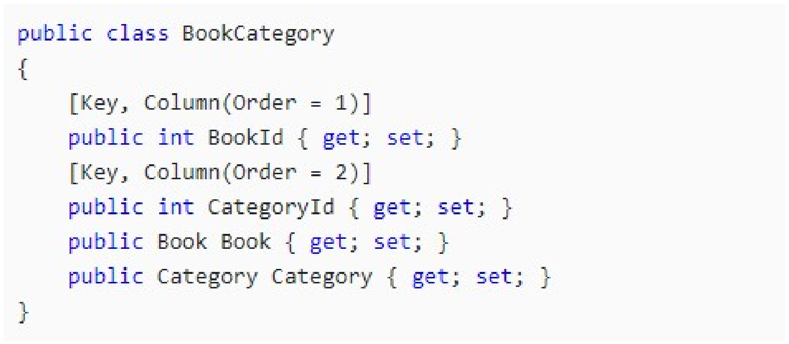
```
CREATE TABLE [Posts] (
    [PostId] int NOT NULL IDENTITY,
    [Title] nvarchar(max) NULL,
    [Content] nvarchar(max) NULL,
    CONSTRAINT [PK_Posts] PRIMARY KEY ([PostId])
);

CREATE TABLE [Tags] (
    [TagId] nvarchar(450) NOT NULL,
    CONSTRAINT [PK_Tags] PRIMARY KEY ([TagId])
);

CREATE TABLE [PostTag] (
    [PostsId] int NOT NULL,
    [TagsId] nvarchar(450) NOT NULL,
    CONSTRAINT [PK_PostTag] PRIMARY KEY ([PostsId], [TagsId]),
    CONSTRAINT [FK_PostTag_Posts_PostsId] FOREIGN KEY ([PostsId]) REFERENCES [Po
    CONSTRAINT [FK_PostTag_Tags_TagsId] FOREIGN KEY ([TagsId]) REFERENCES [Tags]
);
```

M-to-M

- To configure many-to-many relationship Using Data Annotations, you need to create the Join Table in the model.



```
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public virtual ICollection<BookCategory> BookCategories { get; set; }
}

public class Category
{
    public int CategoryId { get; set; }
    public string CategoryName { get; set; }
    public virtual ICollection<BookCategory> BookCategories { get; set; }
}
```



```
public class BookCategory
{
    [Key, Column(Order = 1)]
    public int BookId { get; set; }
    [Key, Column(Order = 2)]
    public int CategoryId { get; set; }
    public Book Book { get; set; }
    public Category Category { get; set; }
}
```