Chapter 14: An introduction to Entity Framework Core

Gill Cleeren

@gillcleeren

Agenda

- Hello Entity Framework Core
- Adding Entity Framework to the application
- Initialization and migration

Hello Entity Framework Core

Entity Framework (EF) Core is a lightweight, extensible, and cross-platform version of the popular Entity Framework data access technology.

Entity Framework Core

- OR Mapper
  - Eliminates need for writing data-access code
- Support for most database engines
  - Also non-relational
- Rewrite of Entity Framework
  - EF6 remains active, different branch
  - Open source
- Code-first only
  - E6 supports database-first and code-first
- Added to a project through a package
- 2.1 update brings back support for lazy loading
  - Was missing up until now!

Entity Framework Core

Class

```
public class Pie
{
    public int PieId { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
}
```

Table

PieId	**Int (PK)**
Name	String
Description	string

SNOWBALL

Adding Entity Framework to the application

Adding the packages

- PM> Install-Package
  - Microsoft.AspNetCore.Identity.EntityFrameworkCore
  - Microsoft.EntityFrameworkCore.SqlServer
  - Microsoft.EntityFrameworkCore.Tools

It starts with a model

- Model is number of entity classes + context
  - Context represents session with database
  - Gives ability to save and query data
- LINQ queries are used to query data
  - If needed, a SQL query can also be passed via EF Core

It starts with a model

```
public class Pie
{
    public int PieId { get; set; }
    public string Name { get; set; }
    public string ShortDescription { get; set; }
    public string LongDescription { get; set; }
    public string AllergyInformation { get; set; }
    public decimal Price { get; set; }
    public string ImageUrl { get; set; }
    ...
    public virtual Category Category { get; set; }
}
```

The model classes in EF Core

- Convention-based entity classes
  - Extra configuration can be passed to overrule or pass extra information
  - Fluent API or data annotations

```
class MyContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Blog>()
                .Property(b => b.Url)
                .IsRequired();
        }
    }
```

The model classes in EF Core

- Convention-based entity classes
  - Extra configuration can be passed to overrule or pass extra information
  - Fluent API or data annotations

```
public class Blog
    {
        public int BlogId { get; set; }
        [Required]
        public string Url { get; set; }
    }
```

The database context

```
public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options)
      : base(options)
    { }

    public DbSet<Pie> Pies { get; set; }
    public DbSet<Category> Categories { get; set; }
}
```

The database context

- Types that will be included
  - All types which are exposed in a DbSet property are included in the model
  - All types in OnModelCreating
  - All types found using navigation properties

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<AuditEntry>();
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

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

    public Blog Blog { get; set; }
}

public class AuditEntry
{
    public int AuditEntryId { get; set; }
    public string Username { get; set; }
    public string Action { get; set; }
}
```

Excluding types

- Using annotations, we can also exclude types from the model

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public BlogMetadata Metadata { get; set; }
}

[NotMapped]
public class BlogMetadata
{
    public DateTime LoadedFromDatabase { get; set; }
}
```

Excluding types

- Also possible via Fluent API

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Ignore<BlogMetadata>();
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public BlogMetadata Metadata { get; set; }
}

public class BlogMetadata
{
    public DateTime LoadedFromDatabase { get; set; }
}
```

Properties

- All get/set properties are included by default
  - Can be overridden using Fluent API or annotations

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    [NotMapped]
    public DateTime LoadedFromDatabase { get; set; }
}
```

Primary Keys

- Convention-based by default
  - Id or <TypeName>Id will be seen as primary key

```
class Car
{
    public string Id { get; set; }

    public string Make { get; set; }
    public string Model { get; set; }
}
```

```
class Car
{
    public string CarId { get; set; }

    public string Make { get; set; }
    public string Model { get; set; }
}
```

Primary Keys

- Can be overridden as well

```
class Car
{
    [Key]
    public string LicensePlate { get; set; }

    public string Make { get; set; }
    public string Model { get; set; }
}
```

```
class MyContext : DbContext
{
    public DbSet<Car> Cars { get; set; }

    protected override void
OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Car>()
            .HasKey(c => c.LicensePlate);
    }
}

class Car
{
    public string LicensePlate { get; set; }

    public string Make { get; set; }
    public string Model { get; set; }
}
```

SNOWBALL

Basic rules

- If CLR type can be null become optional
  - String, int?, byte[]…
- If not, it becomes required by default
  - Int, bool…
- Can be overridden

```
public class Blog
{
    public int BlogId { get; set; }
    [Required]
    public string Url { get; set; }
}
```

Field length

- Database will choose data type for property
  - Basically what the provider will do for us
- Typically defaults to largest possible option
  - nvarchar(max) for string
- Can be overridden

```
public class Blog
{
    public int BlogId { get; set; }
    [MaxLength(500)]
    public string Url { get; set; }
}
```

Relations between types

- Foreign keys are required as well
- EF Core follows some guidelines here
  - Dependent entity
    - This is the entity that contains the foreign key property(s). Sometimes referred to as the 'child' of the relationship
  - Principal entity
    - This is the entity that contains the primary/alternate key property(s). Sometimes referred to as the 'parent' of the relationship
  - Foreign key
    - The property(s) in the dependent entity that is used to store the values of the principal key property that the entity is related to
  - Principal key
    - The property(s) that uniquely identifies the principal entity. This may be the primary key or an alternate key
  - Navigation property
    - A property defined on the principal and/or dependent entity that contains a reference(s) to the related entity(s)
  - Collection navigation property
    - A navigation property that contains references to many related entities
  - Reference navigation property
    - A navigation property that holds a reference to a single related entity
  - Inverse navigation property
    - When discussing a particular navigation property, this term refers to the navigation property on the other end of the relationship

Relations between types

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

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

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

Relations between types

- Relations are convention-based
    - Will typically be created when navigation property is discovered
    - When the type of a property is not a scalar value, it is considered a navigation property
- Typically, these relations are defined on both sides (fully defined)
    - Only works if just one relation can be discovered
    - Other options are possible
        - Can also be done using annotations

Relations between types

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

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

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

Relations using annotations

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

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

    public int BlogForeignKey { get; set; }

    [ForeignKey("BlogForeignKey")]
    public Blog Blog { get; set; }
}
```

Relations using Fluent API

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Post>()
            .HasOne(p => p.Blog)
            .WithMany(b => b.Posts);
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

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

    public Blog Blog { get; set; }
}
```

Creating indexes

- Convention ensures creation of index for each property used in foreign key
- Can also be added using Fluent API

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void
OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasIndex(b => b.Url);
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Mapping to the relational database

- Using table mapping, we can specify which table, column… needs to be queried in the real database
  - Can be done using annotations or fluent API

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder
modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .ToTable("blogs");
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

```
[Table("blogs")]
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Mapping to the relational database

- Also applies for columns

```
public class Blog
{
    [Column("blog_id")]
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder
modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(b => b.BlogId)
            .HasColumnName("blog_id");
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Mapping to the relational database

- Also possible to override the type of column in the database
  - By default, CLR type and provider decide this, based on property type

```
public class Blog
{
    public int BlogId { get; set; }
    [Column(TypeName = "varchar(200)")]
    public string Url { get; set; }
}
```

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder
modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(b => b.Url)
            .HasColumnType("varchar(200)");
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

SNOWBALL

Querying for data

- LINQ is used to query data from the database

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs.ToList();
}
```

```
_appDbContext.Pies.
  Include(c => c.Category).Where(p =>
p.IsPieOfTheWeek);
```

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);
}
```

SNOWBALL

Loading related data

- Based on navigation properties, EF Core will load related entities
- Different options are available
  - 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
    - **Supported in EF Core 2.1!**

Eager loading

- Use Include to load related data
  - If data was already in context, may be retrieved even without Include

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .ToList();
}
```

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .Include(blog => blog.Owner)
        .ToList();
}
```

Eager loading

- Even possible on multiple levels

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
            .ThenInclude(post => post.Author)
        .ToList();
}
```

```
using (var context = new BloggingContext())
{
    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

```
using (var context = new BloggingContext())
{
    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();
}
```

Modifying data

- Context has ChangeTracker
  - All returned entities are tracked by default (can be changed)
  - Any changes made will be recorded
  - Persisted to database when SaveChanges() is called
  - Results in INSERT, UPDATE, DELETE being called on database

Saving new data

```
foreach (var shoppingCartItem in shoppingCartItems)
{
    var orderDetail = new OrderDetail()
    {
        Amount =  shoppingCartItem.Amount,
        PieId = shoppingCartItem.Pie.PieId,
        OrderId = order.OrderId,
        Price = shoppingCartItem.Pie.Price
    };
    _appDbContext.OrderDetails.Add(orderDetail);
}
_appDbContext.SaveChanges();
```

Modifying data

```
using (var db = new BloggingContext())
        {
            var blog = db.Blogs.First();
            blog.Url = "http://sample.com/blog";
            db.SaveChanges();
        }
```

Deleting data

```
using (var db = new BloggingContext())
        {
            var blog = db.Blogs.First();
            db.Blogs.Remove(blog);
            db.SaveChanges();
        }
```

Combining operations

```
using (var db = new BloggingContext())
        {
            db.Blogs.Add(new Blog { Url = "http://sample.com/blog_one" });
            db.Blogs.Add(new Blog { Url = "http://sample.com/blog_two" });

            var firstBlog = db.Blogs.First();
            firstBlog.Url = "";

            var lastBlog = db.Blogs.Last();
            db.Blogs.Remove(lastBlog);

            db.SaveChanges();
        }
```

Working with related data

- When creating related data, saving one will result in saving many

```
using (var context = new BloggingContext())
        {
            var blog = new Blog
            {
                Url = "http://blogs.msdn.com/dotnet",
                Posts = new List<Post>
                {
                    new Post { Title = "Intro to C#" },
                    new Post { Title = "Intro to VB.NET" },
                    new Post { Title = "Intro to F#" }
                }
            };

            context.Blogs.Add(blog);
            context.SaveChanges();
        }
```

App configuration

```
private IConfigurationRoot _configurationRoot;
public Startup(IHostingEnvironment hostingEnvironment)
{
        _configurationRoot = new ConfigurationBuilder()

        .SetBasePath(hostingEnvironment.ContentRootPath)
                                        .AddJsonFile("appsettings.json")
                                        .Build();
}
public void ConfigureServices(IServiceCollection services)
{
        services.AddDbContext<AppDbContext>(options =>
      options.UseSqlServer(_configurationRoot.GetConnectionString
         ("DefaultConnection")));
}
```

DEMO

Adding Entity Framework

Repository using EF Core

App configuration

Database Initialization

Database Initialization

```
using (var scope = host.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    try
    {
        var context = services.GetRequiredService<AppDbContext>();
        DbInitializer.Seed(context);
    }
    catch (Exception)
    {
        //we could log this in a real-world situation
    }
}
```

Database migrations

- Typically done for every change of the model
- Triggered from Package Manager Console
  - Add-Migration <Name>
  - Update-Database

DEMO

Loading initial data

Using migrations

Summary

- EF Core is successor of EF6
- Most features of EF6 have been moved to EF Core

LAB

Exercise 5. Creating a database with EF Core