Entity Framework Core

C# Programming

In this chapter we will learn ...

- What is Entity Framework
- How to use EF Core to access a database

Entity Framework Core

Persisting to a Database

What is Entity Framework Core?

It's an Object Relational Mapper

That means it can save your objects to a data store and retrieve them again

- Requires some work from you
- Leverages LINQ
- Releases you from the need to write SQL
- Allows you to focus on your data entities

Entity Framework (EF) Core - Approaches

There are two approaches when developing with EF

Code First (our approach)

- You write the data entities
- You annotate them with useful information used in validation etc
- You generate and maintain the database using specific EF migration Tools

Database First

- You reverse engineer an existing database or data model
- You Generate Data Entities from the Data Model

The Data Source

We will use SQL Server and the easiest way to do that is with Docker

- Install Docker Desktop (not on an ARM Mac)
- Wait for it to install
- Run the following from a terminal

```
docker run -e "ACCEPT_EULA=Y" -e "MSSQL_SA_PASSWORD=Password99@" -p 1433:1433 -d mcr.microsoft.com/mssql/server:2022-latest
```

- Download and install the latest version of SQL Server Management Studio (SSMS) and connect

Code First - Define your Data

We are developing a Movie Website and we are starting with the Movie class

```
public class Movie {
    public string Title {get;set;} = default;
    public string Director {get;set;} = default;
    public int Year {get;set;} = default;
}
```

Next add Entity Framework to your project

Using the nuget package manager add the following packages to your project

```
Microsoft.EntityFrameworkCore
Microsoft.EntityFrameworkCore.SqlServer // Or SQLite, MySql etc
Microsoft.EntityFrameworkCore.Tools
```

Annotating the Data Classes

Entity Framework needs extra information to help it build a database from our classes

- Primary Keys - Which fields uniquely identify individual objects
- Referential Integrity - Do some fields relate to data in other tables
- Data Integrity - How short or long should a string be, what is the range for an integer?
- Entity naming - What names should be given to tables, columns etc

Primary Key

Below we have added an Id field to act as a unique identifier for a movie

- Key is a data annotation that tells EF to treat Id as an auto generated identifier

```
public class Movie {

    [Key]
    public int Id {get;set;} = default;
    public string Title {get;set;} = default;
    public string Director {get;set;} = default;
    public int Year {get;set;} = default;
}
```

Requires a **using System.ComponentModel.DataAnnotations;**

Required fields

[Required] attribute is used to define required fields

```
public class Movie {

    [Key]
    public int Id {get;set;} = default;
    [Required]
    public string Title {get;set;} = default;
    [Required]
    public string Director {get;set;} = default;
    [Required]
    public int Year {get;set;} = default;
}
```

Other Annotations

What are these annotations doing?

```
[Table("movie")]
public class Movie {
    [Key]
    public int Id { get; set; }
    [Required]
    [StringLength(85)]
    public string Title { get; set; }
    [Required]
    public string Director { get; set; }
    [Display(Name ="Release Year")]
    public int Year { get; set; }
}
```

Database Context

We now need to connect to a database using EF's DbContext class

1. Write a context class that inherits from DbContext
2. Add generic collections for each table to be added to the database
3. Provide the DbContext with a Database connection engine
    - Usually done using Dependency Injection
    - We will hard code it to begin with

Application Database Context

```
public class ApplicationDBContext: DbContext {

    // Never hard code your database connection string in code!
    // We just want to limit the complexity for teaching purposes and show a better way later
    string conn="Data Source=(local);Initial Catalog=Movies;Integrated Security=false;
    User Id=sa;Password=Password99@;TrustServerCertificate=True";

    public ApplicationDBContext() { } // Not normally provided. Allows EF Tools to operate without Dependency Injection
    // Constructor used for Dependency Injection
    public ApplicationDBContext(DbContextOptions<ApplicationDBContext> options):base(options) { }

    public DbSet<Movie> Movies { get; set; } // Create a DBSet for every table you want in the DB


    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) {
        optionsBuilder.UseSqlServer(conn); // Hard Coded connection - Just for now!
    }
}
```

Why don't we want to store the connection string in the code?

Generating the Database

Installing Microsoft.EntityFrameworkCore.Tools earlier on gave access to:

- Package Management Console Tools to perform
  - Data Model change tracking
  - Database Update management

From within Visual Studio select menu:

- Tools->Nuget Packet Manager->Packet Manager Console
- A console will appear

To Install for For VSCode run the following from a terminal

```
dotnet tool install --global dotnet-ef
```

Adding Migrations

We first need to create a migration that will:

- Generate a class that defines the necessary database changes required since our last migration
- Give our migration a name so we can roll back to it

From the Package Manager Console

```
> Add-Migration InitialCreate
```

or from VSCode

```
dotnet ef migrations add InitialCreate
```

Updating the Database

Next we update the database with changes defined in our recent migration

From the Package Manager Console

```
> Update-Database
```

For VSCode run

```
dotnet ef database update
```

DbContext - Adding a new Movie

```
var movie = new Movie {Title = "Star Wars", Year = 1977, Director = "George Lucas"};

var db = new ApplicationDBContext();

db.Movies.Add(movie);

db.SaveChanges();
```

Querying Movies

Use LINQ to query your DBSets

```
var db = new ApplicationDBContext();

var movies = from movie in db.Movies where movie.Year > 1950 select movie;
// var movies = db.Movies.Where(m=>m.Year>1950);

foreach(var movie in movies){
    Console.WriteLine($"{movie.Title} was released in {movie.Year}")
}
```

Updating

```
var movie = db.Movies.Where(m => m.Title.Equals("Star Wars")).First();

if (movie is not null)
{
    movie.Year++;
    db.SaveChanges();
}
```

Deleting

```
var movie = db.Movies.Where(m => m.Title.Equals("Star Wars")).First();

if (movie is not null)
{

    db.Movies.Remove(movie);
    db.SaveChanges();
}
```

Appendix

One to Many Relationships

Imagine a Director class that looked like this

```
public class Director {
    [Key]
    public int Id { get; set; }
    [Required]
    public string Name { get; set; }
    public ICollection<Movie> Movies { get; set; } // Mapping the movies they have directed
}
```

Allowing the Movie class to look like this

```
public class Movie {
    public string Title {get;set;} = default;
    public Director Director {get;set;} = default; // Maps to the Director class
    public int Year {get;set;} = default;
}
```

- Entity Framework will manage these relational mappings
- It Makes UI construction more difficult
    - Director field on a Movie UI needs to be a dropdown list

Retrieving Child entities

By default EF won't retrieve child entities from a query

- Use **Include** to specify the hierarchical child entities to retrieve from the database

```
foreach(var director in db.Directors.Include("Movies"))
{
    Console.WriteLine(director.Name);
    foreach(var movie in director.Movies)
    {
        Console.WriteLine($"\t{movie.Title}");
    }
}
```

In this chapter we learned ...

- What is Entity Framework
- How to use EF Core to access a database