Object/Relational Mapper
Entity Framework
DbContext

.NET

***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.*

HTTPS://DOCS.MICROSOFT.COM/EN-US/EF/CORE/

Object-Relational Mapping

https://en.wikipedia.org/wiki/Object-relational_mapping

Object-relational mapping (ORM, O/RM, and O/R mapping tool) is a programming technique for converting data between incompatible type systems using OOP languages.

In OOP, data-management acts on objects which have ***non-scalar*** values. For example, an address book contains objects that each represent a single person with attributes to hold the person's name, phone number, address, etc.

The address-book entry is treated as a single object by the programming language and it can be referenced by a single variable containing a pointer to the object.

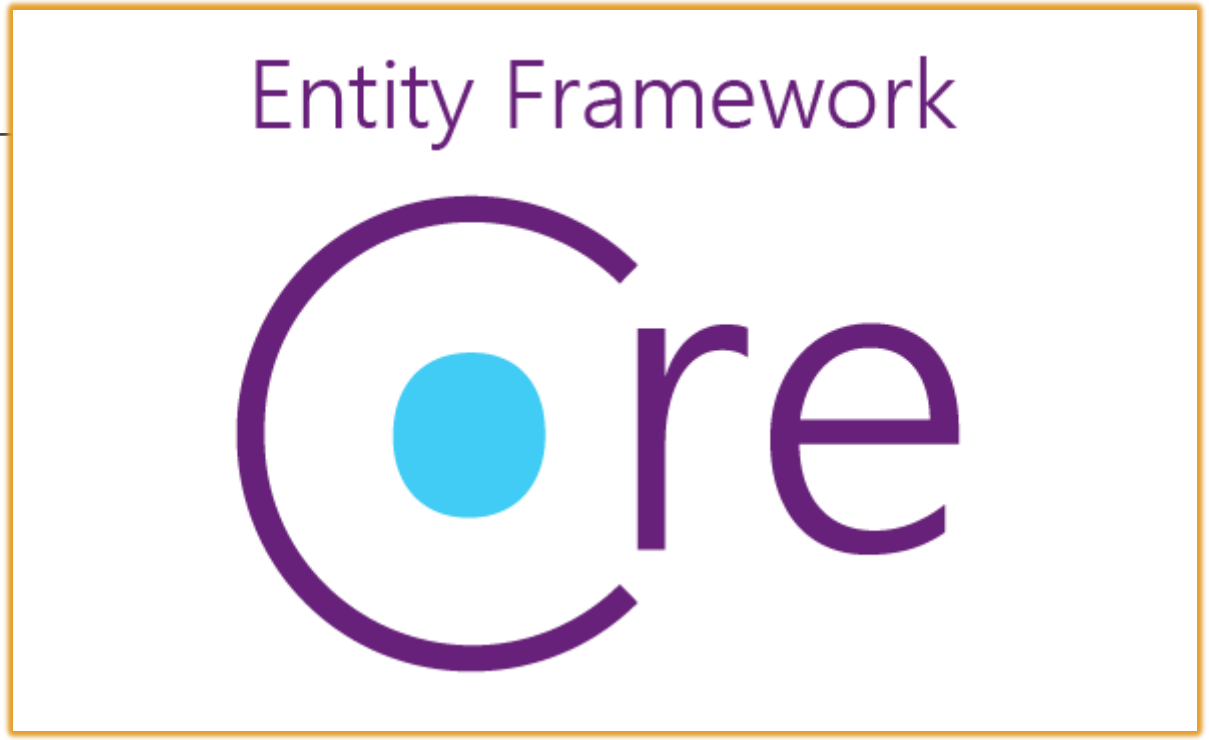
Most DB's can only store and manipulate ***scalar*** values such as integers and strings organized within tables. Object-relational mapping implements a system in which the object values are converted into groups of simpler values for storage in the database and converted back upon retrieval.

The object values must be ***atomic*** to be stored in the database and preserve the properties of the objects and their relationships so that they can be reloaded as objects when needed.
When this functionality is implemented, the objects are said to be **persisted** to the DB.

Name	Age	Pnum	Address
Mark	40	817364	432 M St.
Sally	89	648214	434 M st.

Entity Framework(an O/RM) Overview

https://docs.microsoft.com/en-us/ef/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.

With ***EF Core***, data access is performed using a ***model***. A ***model*** is made up of ***entity classes*** and a context object that represents a session with the database, allowing you to query and save data.

EF Core supports many database engines.

Entity Framework(an O/RM) Overview

https://docs.microsoft.com/en-us/ef/core/#the-model

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.

```
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;

namespace Intro
{
    public class BloggingContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer(
                @"Server=(localdb)\mssqllocaldb;Database=Blogging;Integrated Security=True");
        }
    }

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

Entity Framework(an O/RM) Querying the Context and Saving

https://docs.microsoft.com/en-us/ef/core/#querying

With a DbContext, instances of your entity classes are retrieved from the database using ***Language Integrated Query (LINQ)***.

```
using (var db = new BloggingContext())
{
    var blogs = db.Blogs
        .Where(b => b.Rating > 3)
        .OrderBy(b => b.Url)
        .ToList();
}
```

Data is created, deleted, and modified in the database using instances of your entity classes. ***.SaveChanges()*** is used to persist changes made.

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

Using Entity Framework Code-First Step by Step

https://docs.microsoft.com/en-us/ef/core/get-started/?tabs=netcore-cli

1. Make sure you've downloaded *.NET Core SDK*
2. Create your project.
   - dotnet new console –o [projectName]
3. With *Package Manager Console* (in VS), install the correct package for the EF Core DB Provider you want. (Here is for SQL-Lite)
   - Install-Package Microsoft.EntityFrameworkCore.Sqlite
4. Create the Class models and a class that inherits DbContext *(put* **using Microsoft.EntityFrameworkCore** *at the top of each file.);*
5. *Add a* **DbSet** *for each model you created.*
6. In *Package Manager Console*, install EF Tools
   - Install-Package Microsoft.EntityFrameworkCore.Tools

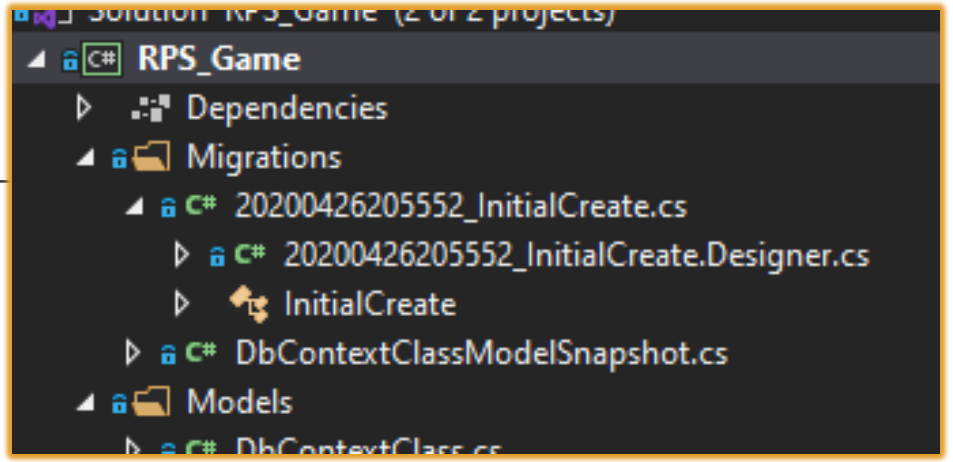
```
public class DbContextClass : DbContext
{
    public DbSet<Game>   Games   { get; set; }
    public DbSet<Round>  Rounds  { get; set; }
    public DbSet<Player> Players { get; set; }

    protected override void OnConfiguring
      (DbContextOptionsBuilder options)
    {
        if(!options.IsConfigured)
        options.UseSqlite("Data Source=blogging.db");
    }
}
```

Using Entity Framework Code-First Step by Step

https://docs.microsoft.com/en-us/ef/core/get-started/?tabs=netcore-cli

1. In Package Manager Console, create the initial set of tables for the model.
   - Add-Migration InitialCreate
2. In Package Manager Console, create the DB and apply the new migration to it.
   - update database
   - Look at the files created to verify that EFCore has correctly interpreted your models.
3. Create a context to use in Main() or in whichever class you need the ***DbContext***.
   - using (var db = new BloggingContext()){ use the context here}
4. Run the app
   - dotnet run




```
namespace RPS_Game.Migrations
{
    [DbContext(typeof(DbContextClass))]
    [Migration("20200426205552_InitialCreate")]
    partial class InitialCreate
    {
        protected override void BuildTargetModel(ModelBuilder modelBuilder)
        {
#pragma warning disable 612, 618
            modelBuilder
                .HasAnnotation("ProductVersion", "5.0.0-preview.3.20181.2");

            modelBuilder.Entity("RPS_Game.Models.Game", b =>
                {
                    b.Property<int>("GameId")
                        .ValueGeneratedOnAdd()
                        .HasColumnType("INTEGER");

                    b.Property<int?>("WinnerPlayerId")
                        .HasColumnType("INTEGER");
```

Migrations – Code First
Create and Update the DB

https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli

---

The ***migrations*** feature in ***EF Core*** provides a way to incrementally update the database schema to keep it in sync with the application's data ***model*** while preserving existing data in the database.

Migrations includes command-line tools and APIs that help with the following tasks:

- Create a ***migration***. Generate code that can update the database to sync it with a set of ***model*** changes.
- Update the database. Apply pending ***migrations*** to update the database ***schema***.
- Customize ***migration*** code. Sometimes the generated code needs to be modified or supplemented.
- Remove a ***migration***. Delete the generated code.
- Revert a ***migration***. Undo the database changes.
- Generate SQL scripts. You might need a script to update a production database or to troubleshoot ***migration*** code.
- Apply ***migrations*** at runtime. When design-time updates and running scripts aren't the best options, call the Migrate() method.

Migrations

https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli

---

After you've defined your initial model, in the command line, create the database with:

- dotnet ef migrations add InitialCreate

Three files are added to your project under the Migrations directory:

- *XXXXXXXXXXXXX_InitialCreate.cs* -- The main migrations file. Contains the operations necessary to apply the migration (in Up()) and to revert it (in Down()).
- *XXXXXXXXXXXXX_InitialCreate.Designer.cs* -- The migrations metadata file. Contains information used by EF.
- *MyContextModelSnapshot.cs* -- A snapshot of your current model. Used to determine what changed when adding the next migration.

The timestamp in the filename helps keep them ordered chronologically so you can see the progression of changes.

After making changes to the model, you will need to update the DB with:

- dotnet ef migrations add AddProductReviews

Using Entity Framework with Sqlite

https://docs.microsoft.com/en-us/ef/core/get-started/?tabs=visual-studio

---

1. Open a console app
2. Install ***Microsoft.EntityFrameworkCore.Sqlite*** (for using the Sqlite DB) or for AzureDB install ***Microsoft.EntityFrameworkCore.***
3. Create a class(es) for the Model(s)
4. Create a class for the ***DbContext***
5. Install Install-Package with ***Microsoft.EntityFrameworkCore.Tools***
6. Use Migrations with ***Add-Migration InitialCreate***
7. Update the DB with the current Models with ***Update-Database*** (you'll do this every time there is a change)
8. In Program.cs, encase the whole program in ***using (var db = new BloggingContext())***
9. Double-Click the project and paste ***<StartWorkingDirectory>$(MSBuildProjectDirectory)</StartWorkingDirectory>*** inside (and at the bottom of) the <PropertyGroup> area.
10. This project shows how to download EFCore and use it with Sqlite.

Entity Framework

https://docs.microsoft.com/en-us/ef/core/

He's building a demo project with Sqlite and Classes for Person and Address. He's got Sqlite working then he adds a column. The old DB needs to be updated. The solution is to delete the file where the DB is held.

1. We created an Azure DB with the lowest level and in the program switched from using .UseSqlite to using .UseSqlServer then we used the connection string provided by Azure to connect. (use the tutorial above)
2. We still needed to set the Azure firewall rule to allow us in. Click on "Set Server Firewall" to configure any computers to use the DB
3. Give a rule name. Get the Client IP address and paste it into both the Start IP and End IP BUT change the final 3 digits to .000 (for Start IP) and .255 (for End IP).
4. CLICK *SAVE*
5. Add the attribute [DatabaseGenerated(DatabaseGeneratedOption.None)] to the Id variable of the models (classes) to prevent SqlServer from forcing it to be the unique id.