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.

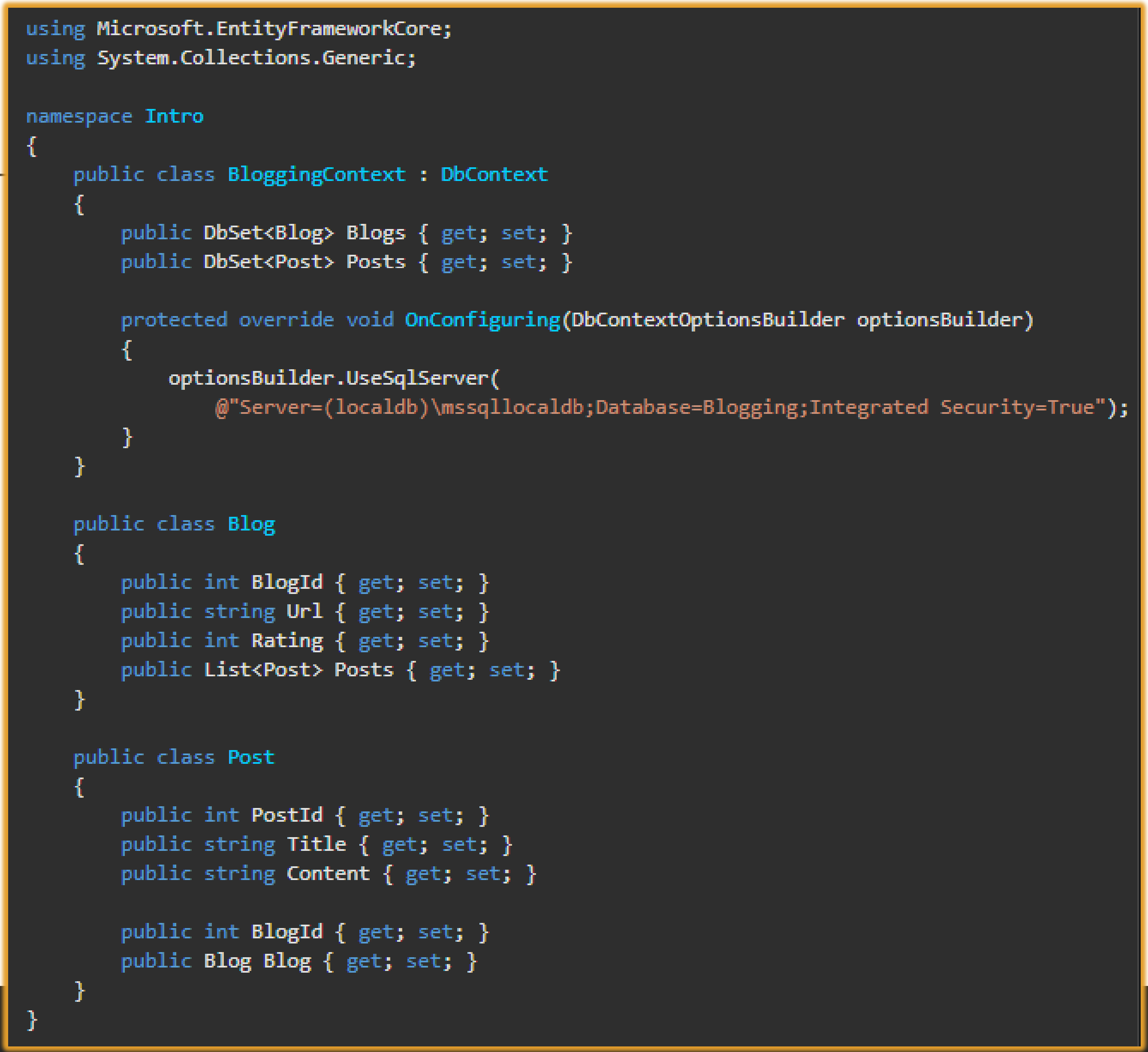
With ***EF Core***, data access is performed using a ***model***. A ***model*** is made up of ***entity classes*** and a context object that represents a session with the database, allowing you to query and save data.

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.

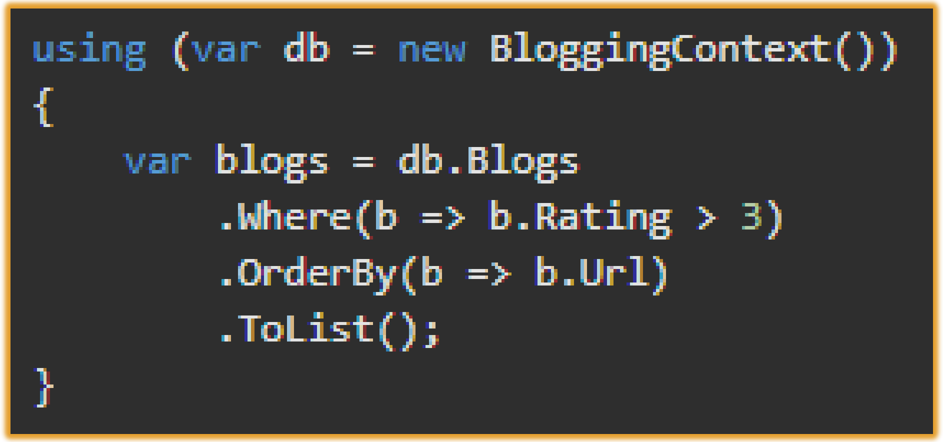


```
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;

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

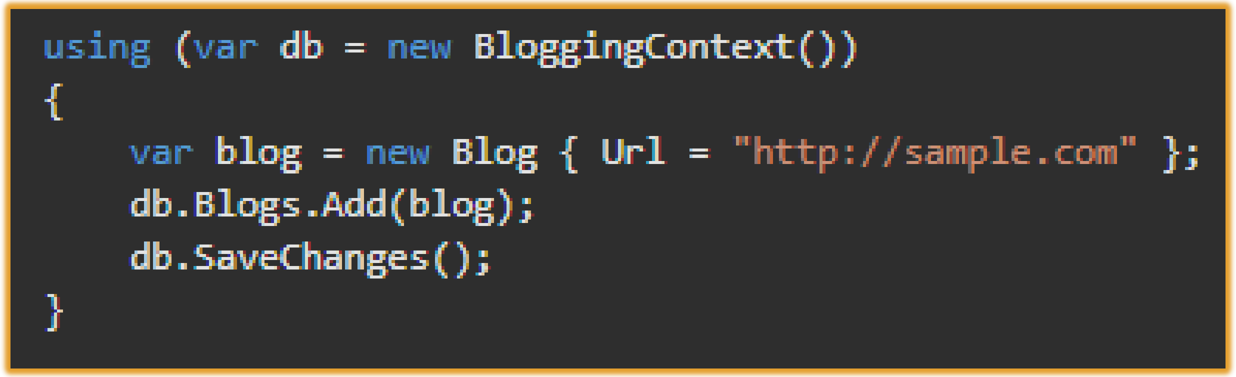
https://docs.microsoft.com/en-us/ef/core/#querying

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. Download .NET Core SDK
2. Create your project.
   - dotnet new console –o [projectName]
3. Install the correct package for the EF Core DB Provider you want. (Here is for SQL-Lite)
   - dotnet add package Microsoft.EntityFrameworkCore.Sqlite
4. Create the program with models and a class that inherits **DbContext**
5. Create a .cs file (or multiple) to hold your DBContext (call it DbContext) Class and models classes
6. Migrations – Install EF
   - dotnet tool install --global dotnet-ef
7. Install EFCore Design (to run the command on a project)
   - dotnet add package Microsoft.EntityFrameworkCore.Design

Using Entity Framework Code-First Step by Step

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

1. Create the initial set of tables for the model.
   - dotnet ef migrations add InitialCreate
2. Create the DB and apply the new migration to it.
   - dotnet ef database update
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

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, create the database with:

- dotnet ef migrations add InitialCreate

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

- *XXXXXXXXXXXXXX_InitialCreate.cs* -- The main migrations file. Contains the operations necessary to apply the migration (in Up()) and to revert it (in Down()).
- *XXXXXXXXXXXXXX_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