Object/Relational Mapper Entity Framework DbContext

.NET

*An **object-relational mapper (O/RM)** enables developers to work with a database using objects that represent the state of the database. This eliminates 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.

A ***scalar*** value is a single number or string value. 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 is referenced by a variable containing a pointer to the object.

fName	lName	Age	Pnum	Address
Dennis	Rader	82	817-364-1234	6220 Independence St.
Joseph	Otero	72	648-214-2345	803 N. Edgemoore St.

Object-Relational Mapping

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

Most DB's can only store and manipulate ***scalar*** (individual/atomic) 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*** (indivisible) 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, data doesn't change between transactions. The objects are said to be persisted to the DB.

fName	lName	Age	Pnum	Address
Dennis	Rader	82	817-364-1234	6220 Independence St.
Joseph	Otero	72	648-214-2345	803 N. Edgemoore St.

Entity Framework(an O/RM)

https://docs.microsoft.com/en-us/ef/core/
https://www.entityframeworktutorial.net/what-is-entityframework.aspx

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

EF Core can serve as an ***object-relational mapper (O/RM).*** It enables .NET developers to work with a database using .NET objects and eliminates the need for most of the data-access code they usually need to write.

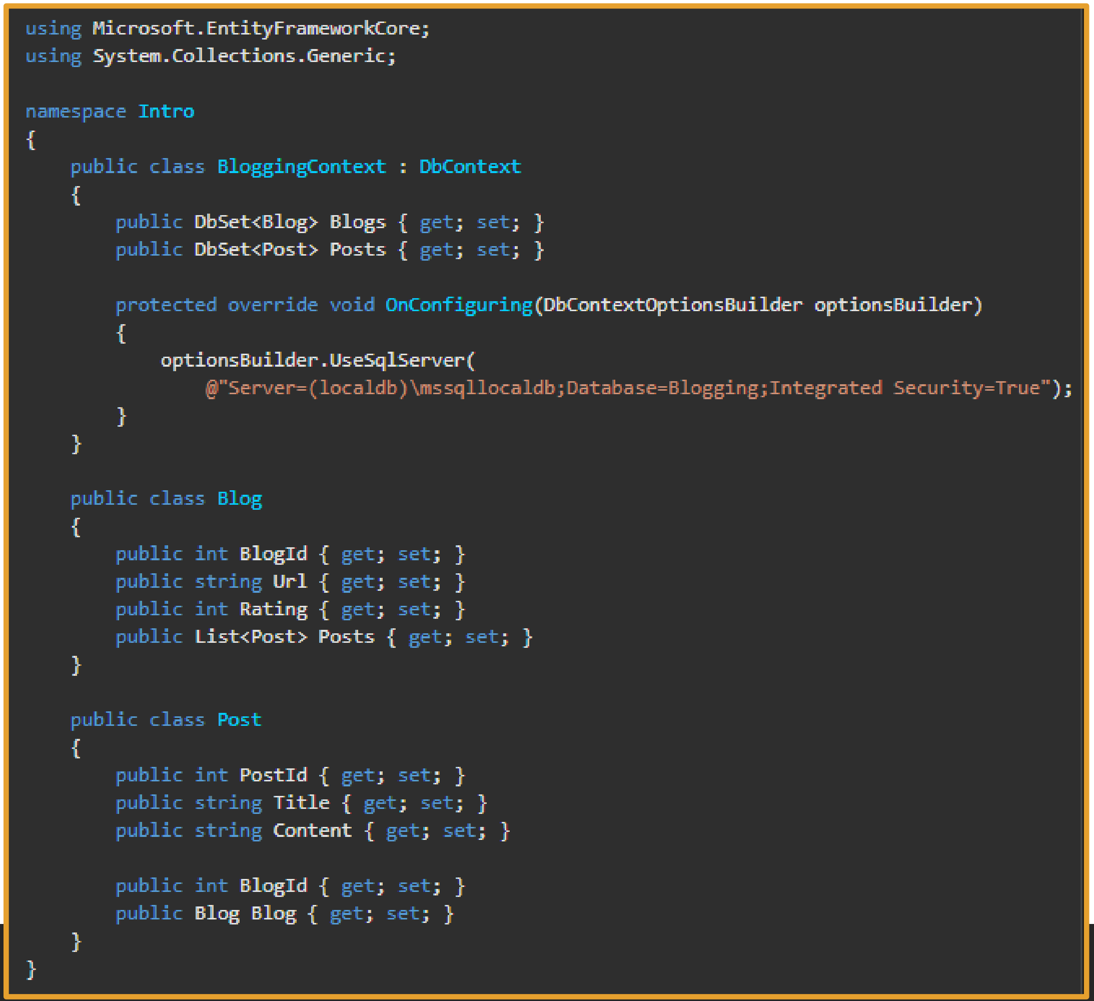
With ***EF Core***, data access is performed using a ***Model***. ***Models*** are made up of ***entity classes*** and a ***context*** object that represents a session with the database, allowing you to query and save data (for example, using ***LINQ***). ***EF Core*** supports many database engines.

Entity Framework - Overview

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

With Entity Framework Core, you can:

- generate a ***Model*** from an existing database (Db-First Approach)
- use ***EF Migrations*** to create a new database from an existing ***Model*** (Code-First Approach)
- hand-code a ***Model*** to match your existing database.



```
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; }
    }
}
```

Fluent API

https://www.entityframeworktutorial.net/efcore/fluent-api-in-entity-framework-core.aspx
https://www.tutorialspoint.com/entity_framework/entity_framework_fluent_api.htm

Entity Framework ***Fluent API*** is used to configure domain classes to override conventions. The ModelBuilder class acts as a Fluent API.

Entity Framework Core Fluent API configures the following aspects of a model:

- Model Configuration: Configures an EF model to database mappings. Configures the default Schema, DB functions, additional data annotation attributes and entities to be excluded from mapping.
- Entity Configuration: Configures entity to table and relationships mapping e.g. PrimaryKey, AlternateKey, Index, table name, one-to-one, one-to-many, many-to-many relationships etc.
- Property Configuration: Configures property to column mapping e.g. column name, default value, nullability, Foreignkey, data type, concurrency column etc.

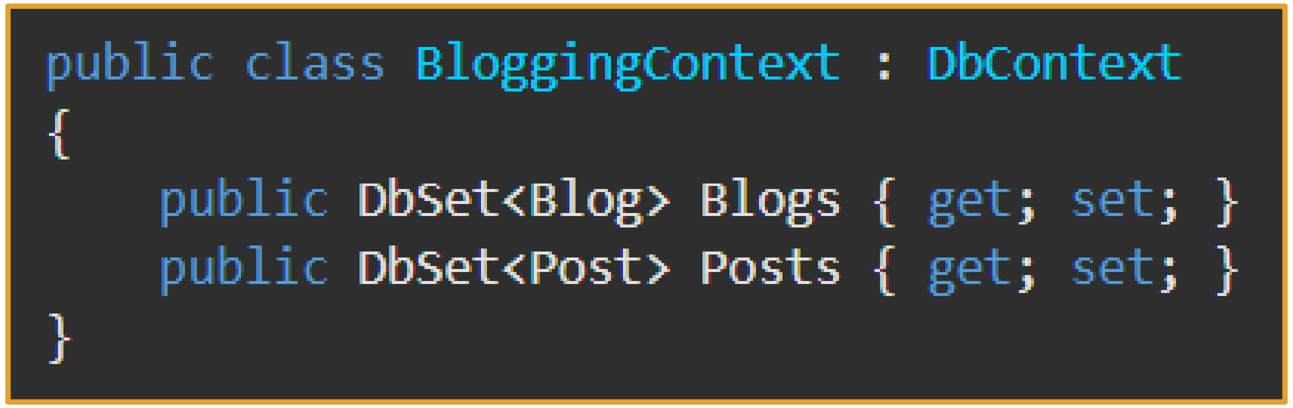
```
public class SchoolDBContext: DbContext
{
    public DbSet<Student> Students { get; set; }

    protected override void
OnModelCreating(ModelBuilder modelBuilder)
    {
        //Property Configurations
        modelBuilder.Entity<Student>()
                .Property(s => s.StudentId)
                .HasColumnName("Id")
                .HasDefaultValue(0)
                .IsRequired();
    }
}
```

Code First Approach - DbSet<>

https://docs.microsoft.com/en-us/ef/ef6/modeling/code-first/dbsets

When using an EF Core ***Code First*** approach, you define a ***DbContext*** that represents your session with the database and exposes (creates) a ***DbSet<modelType>*** for each Model ***type*** in your application.

This will configure the Classes sent as ***type*** arguments to the ***DbSet<>*** Class as entity ***types*** in your DB, as well as automatically configuring other needed ***types*** reachable from those ***types***.



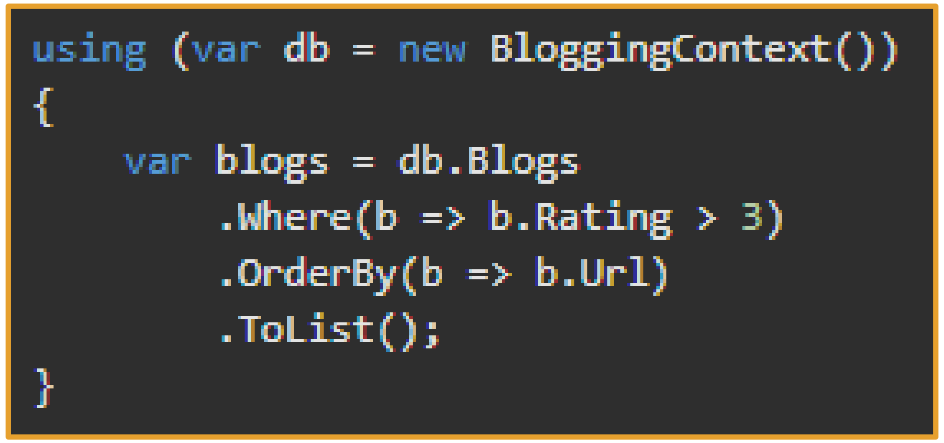
```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
}
```

Querying DbContext / .SaveChanges();

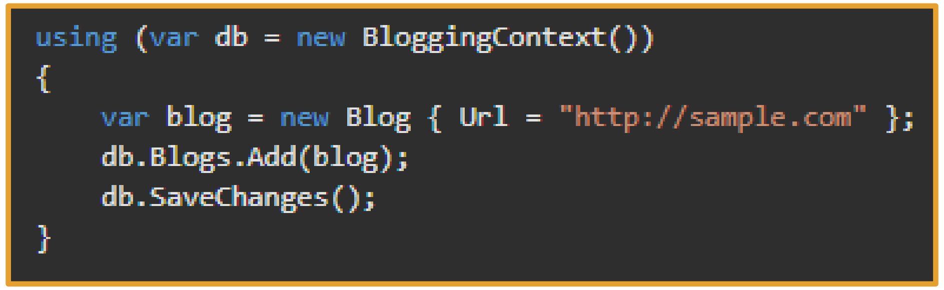
https://docs.microsoft.com/en-us/ef/core/#querying
https://www.entityframeworktutorial.net/efcore/entity-framework-core-dbcontext.aspx

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

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



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



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

Migrations – Code First Approach 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.

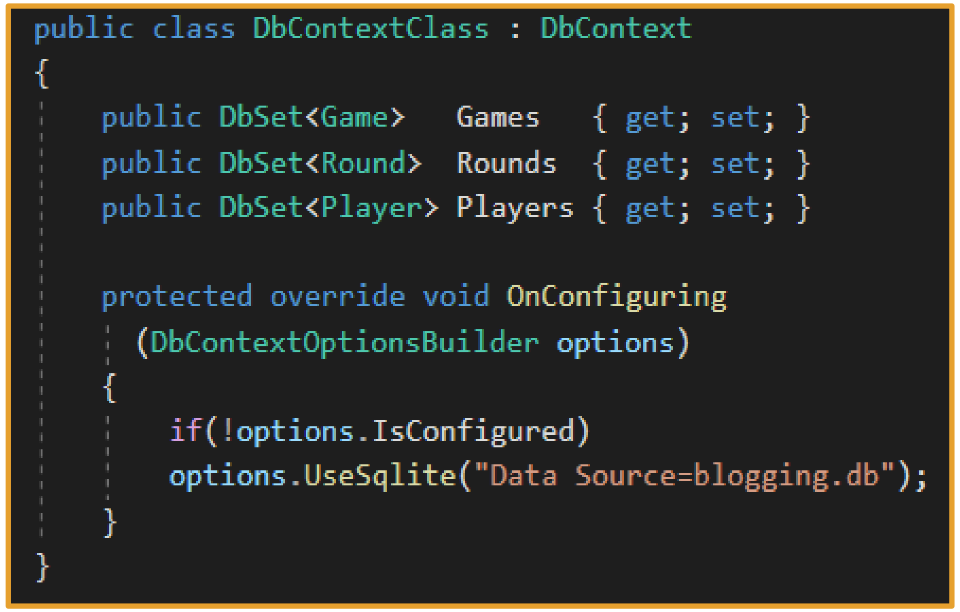
Migrations includes command-line tools and APIs that help with the following tasks:

- 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.

EF Code-First with Sqlite (1/2)

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

1. Create your console project with dotnet cli (below) or with Visual Studio.
   - dotnet new console –o [projectName]
2. 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
3. Create Class models that represent the entities in your application.
4. Create a class that inherits DbContext, add DbSet<>'s, and make it match the image to the right.
5. In ***Package Manager Console***, install EF Tools
   - Install-Package Microsoft.EntityFrameworkCore.Tools

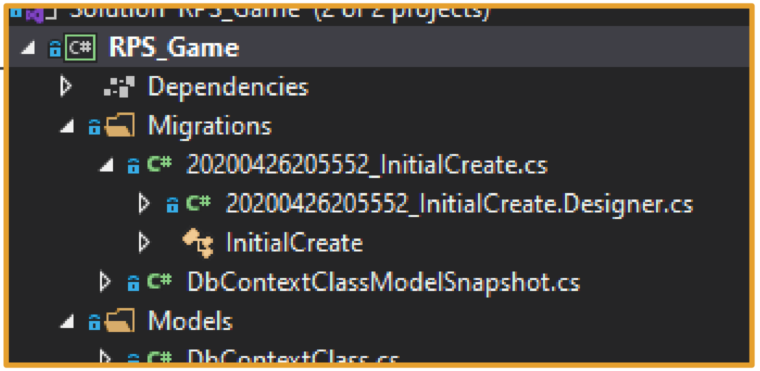


```
public class DbContextClass : DbContext
{
    public DbSet<Game>   Games   { get; set; }
    public DbSet<Round>  Rounds  { get; set; }
    public DbSet<Player> Players { get; set; }

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

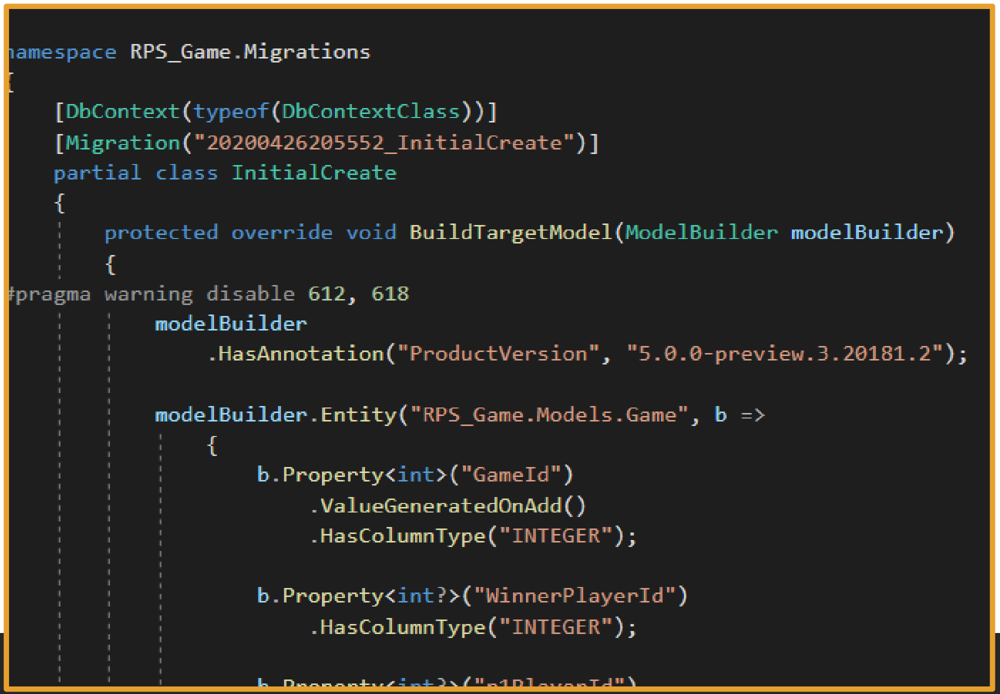
EF Code-First with Sqlite (2/2)

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

1. Create the initial set of migration instructions files for Db creation.
   - Add-Migration InitialCreate
2. 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 whichever class you need the ***DbContext***.
   - var db = new BloggingContext();



```
Solution 'RPS_Game' (2 of 2 projects)
  RPS_Game
    Dependencies
    Migrations
      20200426205552_InitialCreate.cs
        20200426205552_InitialCreate.Designer.cs
        InitialCreate
      DbContextClassModelSnapshot.cs
    Models
      DbContextClass.cs
```



```
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");

                    b.Property<int?>("p1PlayerId")
```

EF Data-First Approach

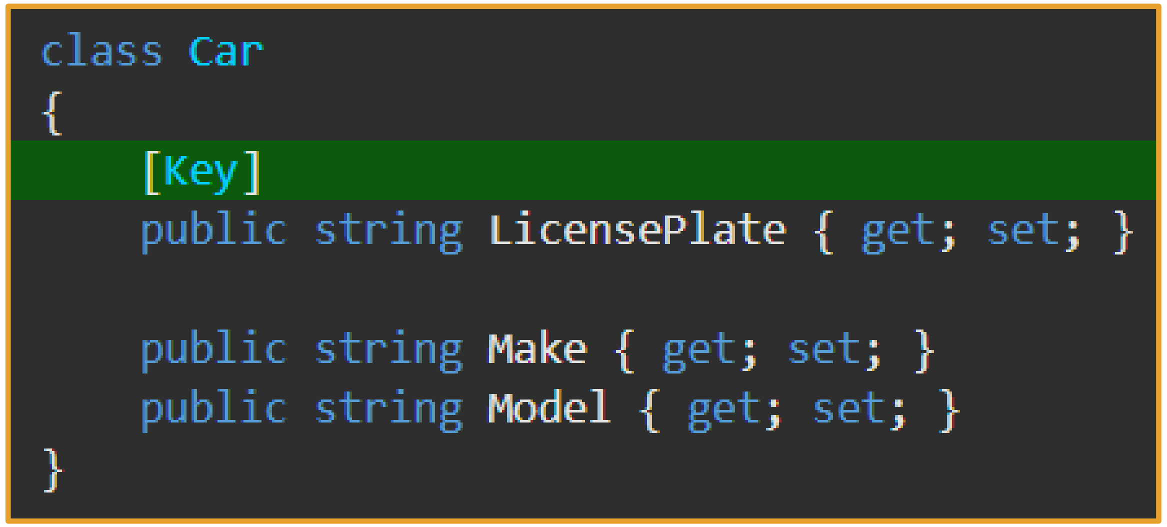
https://www.entityframeworktutorial.net/efcore/create-model-for-existing-database-in-ef-core.aspx

- Download the required packages.
  - Microsoft.EntityFrameWorkCore.SqlServer
  - Microsoft.EntityFrameWorkCore.Design
  - Microsoft.EntityFrameWorkCore.Tools
- In VS, go to the Package Manager Console and enter:
  - Scaffold-DbContext "Server=.\SQLExpress;Database=YourDbNameHere;Trusted_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
- /

Database - Entity Framework [key]

https://docs.microsoft.com/en-us/ef/core/modeling/keys?tabs=data-annotations

A ***key*** serves as a unique identifier for each ***entity*** instance. Most ***entities*** in ***EF*** have one ***key*** which maps to the concept of a ***Primary Key*** in relational databases. It is possible for an ***entity*** to have no keys. ***Entities*** can also have additional ***keys*** (Alternate ***Keys***) beyond the ***Primary Key***. By default, any property named Id or <type name>Id will be automatically configured by ***EF*** as the ***Primary Key*** of an ***entity***.



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

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

You can force configure any single property to be the Primary Key of an entity.