Școala
informală
de IT

EF Core

ORM

ORM

- Object-Relational Mapping (ORM) is a programming technique for automatic mapping data and database schema
    - Map relational DB tables to classes and objects
- ORM creates a "virtual object database"
    - Used from the programming language (C#, Java, PHP, …)
- ORM frameworks automate the ORM process
    - A.k.a. Object-Relational Persistence Frameworks

EF Core

EF Core

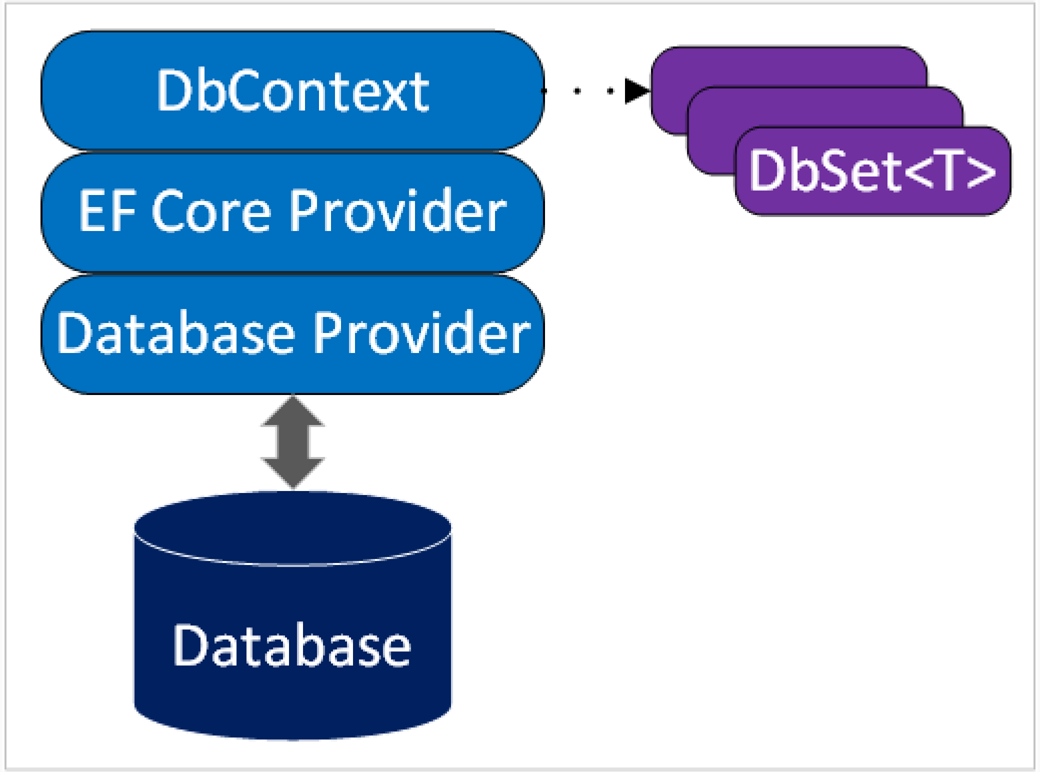
- Entity Framework (EF) is the standard ORM framework for .NET
  - Maps relational database to C# object model
  - Abstracts the underlying data provider
  - Powerful data manipulation API over the mapped schema
  - CRUD operations and complex querying with LINQ
  - Installed via Nuget
  - Cross -platform

Features

- Works with a variety of database servers (Microsoft SQL server, Oracle, SQLite, PosgreSQL)
- Rich mapping engine handle real-world database and work with stored procedure
- Generates strongly typed entity objects that can be customized beyond 1-1 mapping
- Generates mapping/plumbing code
- Translates LINQ queries to database queries
- Materializes objects from data store calls
- Tracks changes, generating updates/inserts

Benefits

- Reduced development time
- Abstracts underlying database(it doesn’t matter what it is)
- Free from hard-coded dependencies on a particular data engine
- Mappings can be changed without changing the application code


DbContext
EF Core Provider
Database Provider
DbSet<T>
Database

EF components

Database providers

- SqlServer
- SqlLite
- In-Memory
- 3rd party

DBContext class

- Represents a session with the database
- Needs to be extended by your own context class
- Implements a combination of Repository and UnitOfWork pattern
- It’s like your own database in code holding a collection of

  DbSet<T>

DBContext class

- Your own context class allows you to:
  - customize model creation
  - seed the database with initial data

- Provides model configuration through FluentApi
- Configured through DbContextOptions/DbContextOptionsBuilder

Your own context example

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

    public DbSet<Car> Car{ get; set; }
}


Dependency Injection

public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<CarContext>(options => options.UseSqlite("Data
Source=car.db"));
}
```

Your own context example(2)

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

    protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
    {
        optionsBuilder.UseSqlite("Data Source=cars.db");
    }
}
```

Your own context example(2)

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

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlite("Data Source=cars.db");
    }
}
```

DBSet<T>

- Represents a collection of entities of a given type
- Used to query and save instances of entities
- Linq statements against a DbSet are translated into queries against the data store
- Items changed, removed or added are not persisted until SaveChanges() is called

Connection Strings

- To be able to communicate with a database you need to connect to a real database(either local or remote)

```
{
"ConnectionStrings": {
    "ContosoPets": “ConnectionString"
  }
}
```

Connection Strings

- Connection string format

```
Data Source=<SQL-server-name>.database.windows.net;Initial
Catalog=ContosoPets;Connect
Timeout=30;Encrypt=True;TrustServerCertificate=False;ApplicationIntent=ReadWrite;
MultiSubnetFailover=False
```

Connection Strings

- In Startup.cs

    In ConfigureServices method

```
services.AddDbContext<PetsContext>(options =>
{
    options.UseSqlServer(Configuration.GetConnectionString("ContosoPets")
});
```

Entities

Entities

- POCO classes
- Mapped to relational data trough configuration
- Relate to other entities trough navigation properties
- Can have annotations
- If a database-first approach is used -> generated as partial classes

Entities or Domain Objects

- EF uses a set of conventions to build a model based on entity classes
- You can use Data annotation to override or hydrate entity classes or Fluent API
- Fluent API has highest precedence and will override conventions and annotations

Model with Fluent API

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

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

Model with Data Annotations

```
public class Car
 {
     public int CarId{ get; set; }
     [Required]
     public string LicensePlate{ get; set; }
 }
```

Conventions

- A property of a class named Id will be considered the primary key

```
class Car
{
    public string Id { get; set; }
    public string Make { get; set; }
    public string Model { get; set; }
    public string LicensePlate{ get; set; }
}
```

- You can override this on OnModelCreating or with Annotations

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

Working with data

Creating New Data

- To create a new database row use the method Add(…) of the corresponding collection:

```
var car= new Car()
{
   Make= "Ford",
   LicencePlate= “HKLM”
};

context.Cars.Add(car);
context.SaveChanges();
```

Create a new car object

Mark the object for inserting

This will execute an SQL INSERT

- SaveChanges() sends the data into the database

Updating Data

- To create a new database row use the method **Add(…)** of the corresponding collection:

```
Car car= carContext.Cars.Find(1);
car.Make = "Toyota";
context.SaveChanges();
```

This will execute an SQL SELECT to load the first order

Deleting Data

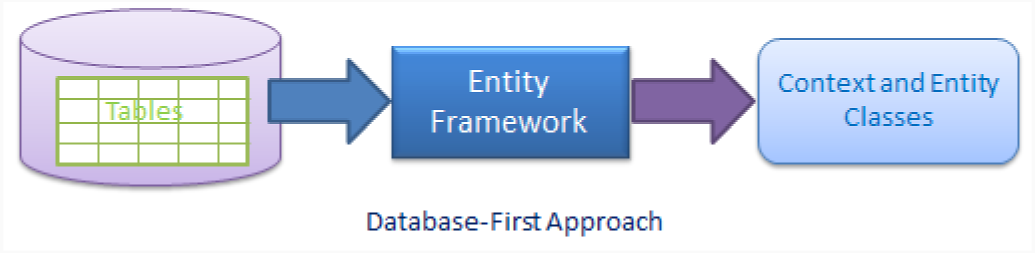
- Delete is done by **Remove()** on the specified entity collection

```
Car car= carContext.Cars.Find(1);
carContext.Cars.Remove(car);
carContext.SaveChanges();
```

Mark the entity for deleting at the next save

execute the SQL DELETE command

Database First Approach


Tables
Entity Framework
Context and Entity Classes
Database-First Approach

Scaffolding

Scaffolding – from CLI

- dotnet ef dbcontext scaffold [datasource][dbprovider] --context <Name> --data-annotations --output-dir <Path>

```
dotnet ef dbcontext scaffold
“Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=T
rue;” Microsoft.EntityFrameworkCore.SqlServer --context CarContext --
data-annotations --force  --output-dir Data/Entities
```

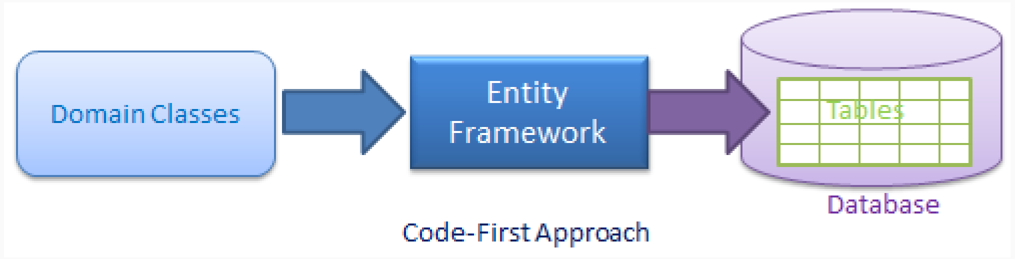
Scaffolding - PowerShell

```
Scaffold-Dbcontext -provider
Microsoft.EntityFrameworkCore.SqlServer -connection
"Server=(localdb)\MSSQLLocalDB;Database=EFCoreMigration;
Trusted_Connection=True;” -OutputDir Models
```

Argument	What it does
-Connection	Required. The connection string to the database.
-Provider	Required. The provider to use. (for example, Microsoft. EntityFrameworkCore. SqlServer
-OutputDir	The directory to put files in. Paths are relative to the project directory.
-ContextDir	The directory to put DbContext file in. Paths are relative to the project directory.
-Context	The name of the DbContext to generate.
-Schemas	The schemas of tables to generate entity types for.
-Tables	The tables to generate entity types for.
-DataAnnotations	Use attributes to configure the model (where possible). If omitted, only the fluent API is used
-UseDatabaseNames	Use table and column names directly from the database.
-Force	Overwrite existing files.

Code-First Approach

Code - First


Domain Classes
Entity Framework
Tables
Database
Code-First Approach

Migrations

Migrations - for existing project

- Use migrations to apply changes from code
- To create the database from the existing code
- To apply code changes - > to the database
- You use the cli

First Migration

```
dotnet ef migrations add InitialCreate \ --project {projectPath}  --context {contextName}
```

```
dotnet ef migrations add InitialCreate \ --project
../ContosoPets.DataAccess/ContosoPets.DataAccess.csproj \ --context ContosoPetsContext
```

Adding a migration

```
dotnet ef migrations add InitialCreate --project {projectPath}  --context {contextName}
```

```
dotnet ef migrations add InitialCreate  --project
../ContosoPets.DataAccess/ContosoPets.DataAccess.csproj \ --context ContosoPetsContext
```

Apply the migration to the database

```
dotnet ef migrations script {lastgeneratedscript} --project {projectPath}
dotnet ef migrations script  Initial  --project Persons
```