Laborator 2
LINQ & Entity Framework

Cate ceva despre **LINQ** = **L**anguage **IN**tegrated **Q**uery (despre asta nu am zis la laborator, dar daca cineva are nevoie de lamuriri am zis sa adaug aici)
Practic, este o sintaxa de query in limbajul de C#.
Query = cerere (in general, de la o colectie, fie ea din baza de date sau din memorie)

SQL Queries:

```
SELECT FirstName
FROM Persons
WHERE IsDeleted = 0;
```

LINQ Queries: (Method syntax)

```
var persons = db.Persons
     .Where(x => x.IsDeleted = 0)
     .Select(x => x.FirstName);
```

In mare, sintaxa de LINQ este una naturala. De exemplu, in SQL desi operatia de SELECT este prima scrisa, ea este ultima executata. In LINQ operatiile se intampla fix in ordinea in care sunt scrise metodele, asa ca nu mai e nevoie sa incepem cu fundul 😀.

In C# / .NET, o sa folosim LINQ (method syntax, cea folosita pe metode: Where(), Select(), etc; mai exista si query syntax, dar nu prea mai este folosita) pentru orice fel de cerere. Nu o sa scriem cod de SQL pentru a lua date din baza de date, pentru ca vom accesa prin LINQ Queries.

Ca sa ne obisnuim cu sintaxa putin, sa vedem 2-3 exemple:

Avem urmatoarea lista:

```
string[] words
   = { "hello", "wonderful", "LINQ", "beautiful", "world", "web" };
```

- Cum facem sa luam cuvintele cu cel putin 5 litere?

  pai luam words-urile unde Lungimea <= 5:

  ```
  var shortWords = words.Where(x => x.Length <= 5);
  ```

- Cum facem daca vrem sa luam doar prima litera a lor?

  pai selectam prima litera (x[0])

  ```
  var firstLetters = words.Where(x => x.Length <= 5)
      .Select(x => x[0]);
  ```

- In firstLetters, in momentul asta, avem de doua ori litera w (din world si web).

  Cum facem daca vrem sa luam lierele dinstince?

  pai le selectam Distinct

  ```
  var distinctLetters = words.Where(x => x.Length <= 5)
      .Select(x => x[0])
      .Distinct();
  ```

Dupa cum putem vedea din Where si Select, metodele din LINQ primesc un predicat:

x => x[0]; <- traducere: avem x si din asta rezulta prima litera din x (string => char)

x => x.Length <= 5; <- traducere: avem x si din asta rezulta daca x are lungimea <= 5 (string => bool); Where-ul primeste un bool care ii zice daca entry-ul respectiv trebuie este returnat.

Now, let's do some exercises. :)

Avem clasele:

Author (id, name)

Book (id, title, authorId)

Pentru care facem cate o lista locala in Main si creem niste date mockuite.

1) Cate carti a scris fiecare autor? (aici vom folosi .GroupBy)
2) Afisati detaliile despre fiecare Carte (Id, Title, AuthorName) (metoda .Join)
3) Sortati random Cartile. (OrderBy)

Rezolvari:

1)

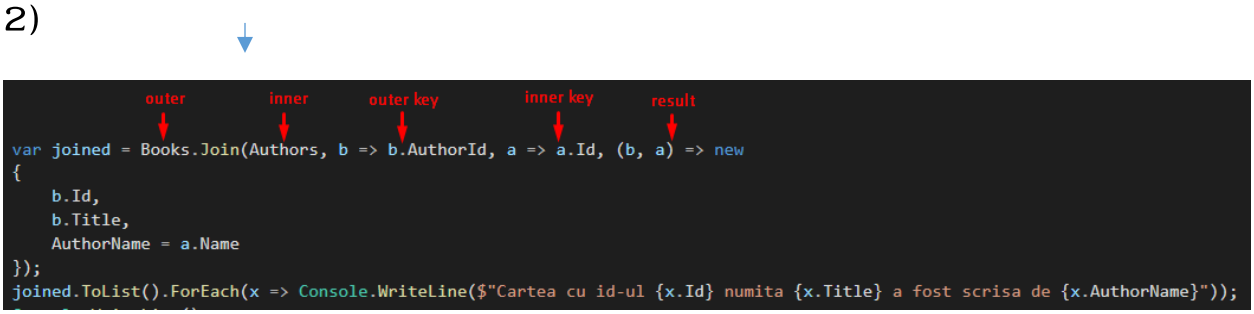
```
var grouped = Books.GroupBy(x => x.AuthorId);
grouped.ToList().ForEach(x =>
      Console.WriteLine($"Autorul {x.Key} a scris {x.Count()} carti"));
```

Aici tot ce de facut este sa apelam metoda GroupBy, transmitandu-i field-ul dupa care vrem sa grupam (in cazul nostru, AuthorId).

Rezultatul metodei GroupBy este o lista de elemente grupate (Grouping).

Acest Grouping nu este mai mult decat o lista cu o proprietate in plus (cea de Key).

2)



Practic, se traduce in SQL:

```
SELECT b.Id, b.Title, a.Name as AuthorName
FROM Books b
      INNER JOIN Author a ON b.AuthorId = a.Id;
```

outer JOIN inner ON outerKey = innerKey

Result este putin mai special:

```
(b, a) => new
{
       b.Id,
       b.Title,
       AuthorName = a.Name
}
```

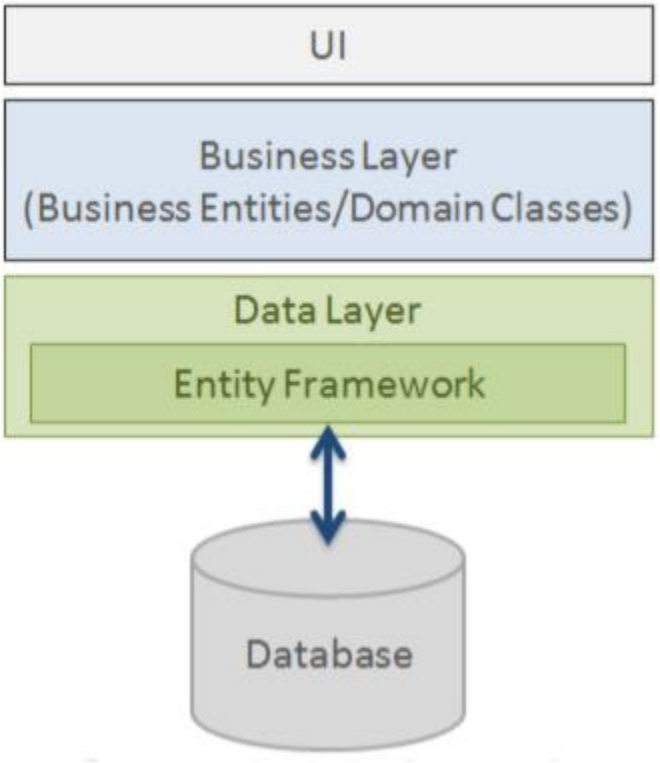
Se traduce: din b si a (ordinea este outer, inner; b si a sunt denumirile date local, nu sunt vazute in afara predicatului) rezulta un nou obiect (in cazul nostru, un obiect anonym – nu are un anumit tip creat).

3)

```
var randomOrder = Books.OrderBy(x => Guid.NewGuid());
randomOrder.ToList().ForEach(x => Console.WriteLine(x.Title));
```

Este o metoda mai dubioasa, but it's working. Se bazeaza pe cat cate de aleatory pare un obiect de tipul Guid (ex: e9c9e9c8 - 602c - 42d7 - 8ec8 - 2beb129a2aac / 02d8b4ee-a019-4ad8-a923-8fdf6f44221d). Ideea e cam asa: fiecarui x i se atribuie un nou Guid, pe care il foloseste in compararea pentru ordonare.

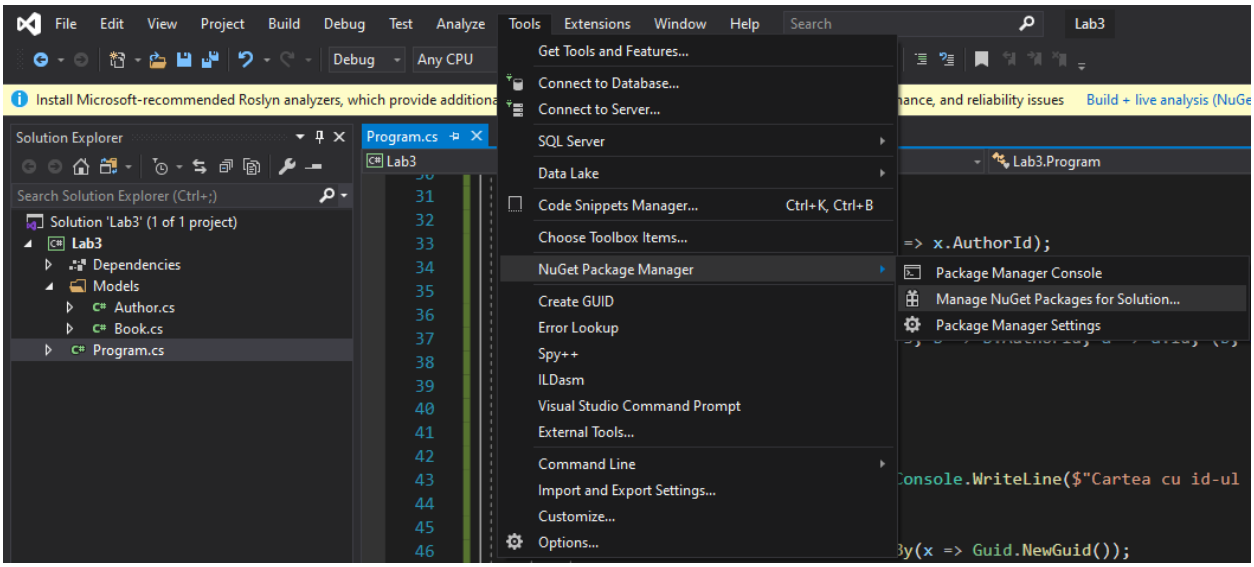
Entity Framework – este un ORM (object-relation mapper) ce leaga obiectele din codul de .NET de obiectele din baza de date. Cu alte cuvinte, el permite dezvoltatorilor de aplicatii sa controleze datele din baza de date prin intermediul codului scris



Dupa cum se vede în figura de mai sus, entity framework functioneaza creand un layer în plus între clasele scrise în cod si baza de date. El se ocupa cu convertirea obiectelor de la un layer la celalalt. EF Core vine cu 2 posibile abordari: code-first si database-first. Cea mai populara viziune este prima, cea bazata pe cod. Astfel, framework-ul creeaza baza de date, tabelele si relatiile dintre ele pe baza codului scris dinainte; de aici denumirea: code-first = mai întâi cod; acelasi principiu de denumire este si la database-first = mai întâi baza de date: este creata la început baza de date,

iar modelele si contextul din cod sunt generate de catre framework pe baza a ce este deja scris.

Pentru a respecta conventiile de modularizare a .NET Core-ului, EF Core vine, impartit sub forma de pachete care trebuiesc instalate prin NuGet Package Manager.



Pentru a instala Entity Framework, o sa fie nevoie sa instalam 4 pachete prin interfata de NuGet:

- EntityFrameworkCore, care este ORM-ul in sine
- EntityFrameworkCore.Relational
- EntityFrameworkCore.SqlServer, care este provider-ul spre SqlServer (baza de date pe care o vom folosi)
- EntityFrameworkCore.Tools, care contine comenzile de rulare petru consola.

Toate 4 pot fi cautate si instalate din interfata de mai sus, tab-ul de Browse.
(Selectezi pachetul droit si apesi install, is that simple)
Primul pas in utilizarea EF-ului este acela de a crea a clasa de tip **context**. Aceasta este cea mai importanta piesa, intrucat poate fi vazuta ca o sesiune a bazei de date de lucru. Este, pracitc, o oglinda a acesteia, din cod. Ea nu este mai mult decat o

clasa normala, ce mosteneste clasa DbContext pe care ne-o furnizeaza EF-ul. Aici vom specifica set-urile dedate, corespondente cu tabele din baza de date, sub forma unor liste DbSet<Object>. DbContext este, astfel, clasa care leaga modelele de EF pentru a putea fi transpuse in baza de date. De asemenea, in clasa DbContext vom avea si metoda ce configureaza Contextul, OnConfiguring, in care vom specifica ConnectionString-ul spre baza de date pe care o vom folosi.

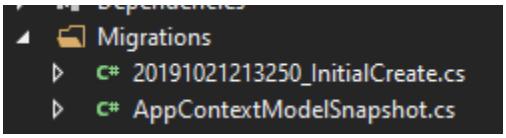
```
0 references
public class AppContext : DbContext
{
    0 references
    DbSet<Author> Authors { get; set; }
    0 references
    DbSet<Book> Books { get; set; }

    0 references
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder
            .UseSqlServer(@"Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=database_test;");//ConnectionString
    }
}
```

Dupa ce avem clasa Context cu tot ce ne intereseaza, trebuie sa comunicam framework-ului sa updateze baza de date in functie de ce am scris. Modul de comunicare a EF-ului este de a migra codul; el creeaza o migratie noua de fiecare data cand ii cerem, migratie in care adauga toate schimbarile aferente claselor gasite in DbContext. Prin urmare, va trebui sa il instruim in a crea o noua migrare. Tot ce trebuie sa facem e sa deschidem Package Manager Console (Tools -> NuGet Package Manager -> Package Manager Console) si sa rulam comanda:

- Add-Migration InitialCreate

Dupa ce comanda va fi executata, vom vedea ca a fost creat un nou folder “Migrations”, ce contine doua fisiere:



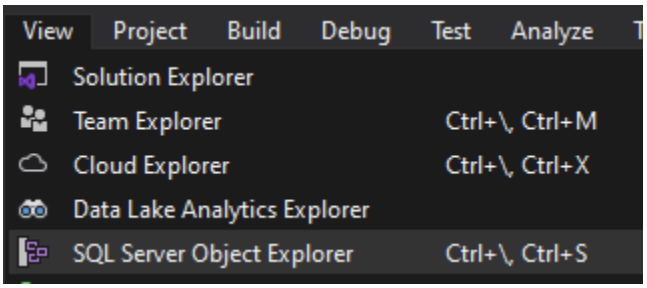
Primul este fisierul cu modificarile (denumit cu numele migrarii pe care I l-am dat noi la creare, caruia i-a fost adaugat si un id bazat pe DateTime-ul curent. Id-ul este folosit in baza de date, intrucat, pentru a tine minte ce migrari au fost rulate pe baza respectiva, EF-ul isi creaza un tabel aditional, numit "__MIgrationHistory" unde stocheaza id-urile migratiilor rulate; astfel, el stie sa aplice doar migratiile noi.
Al doilea fisier, ModelSnapshot, este modalitatea framework-ului de a sti cum arata baza de date la ultima migrare. Practic, este o copie in functie de care, la adaugarea unei noi migratii isi da seama care sunt modificarile fata de starea precedenta.

Acum, avem migrarile create, dar trebuie sa instruim framework-ul in a aplica aceste migratii peste baza de date. Pentru asta avem alta comanda:

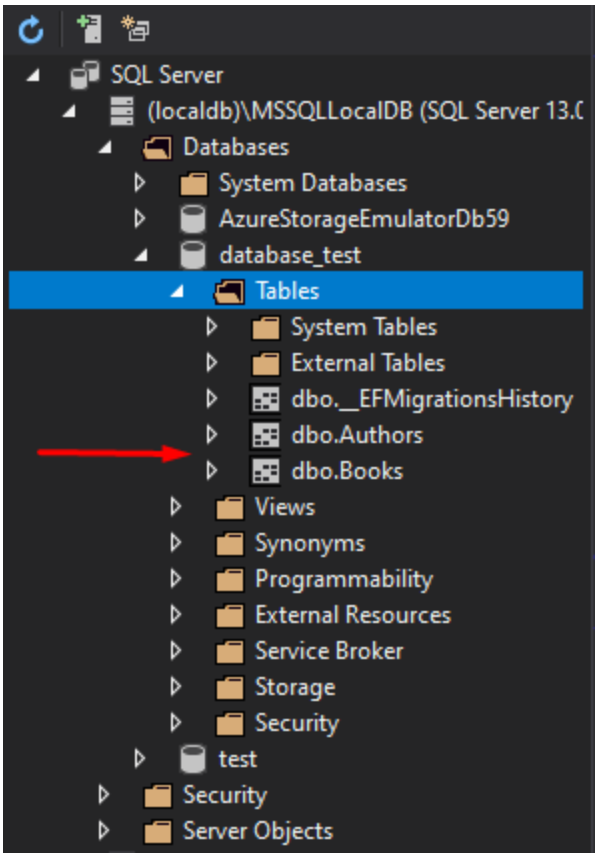
- Update-Database

Astfel, Framework-ul o sa ia migrarile pe care nu le gaseste deja in tabelul de care am zis mai devreme, traduce migrarea in cod SQL (prin intermediul provider-ului de SqlServer pe care l-am instalat mai devreme).

Pentru a verifica ca aceste schimbari intr-adevar s-au realizat, putem merge la View -> Sql Server Object Explorer, sau, recomandarea mea: prin SSMS (instalat din tutorialul de mai jos).



Aici putem vedea exact baza de date. Daca totul a mers bine, aici ar trebui sa vedem modificarile pe care le-am facut in cod.



Pentru a instala SqlServer, necesar pentru a putea avea baza de date, cel mai bine ar fi sa mergem pe urmele tutorialului:
https://www.sqlshack.com/step-by-step-installation-of-sql-server-2017/

Ca sa vorbim putin si de relatii (deoarece folosim baze de date **relationale**):
Avem cele 3 relatii cunoscute de la baze de date: One-To-One, One-To-Many si Many-To-Many; Rezolvarea este prin compunerea de clase.

One-To-One

E destul de basic: trebuie doar sa ii zicem ca clasa X are o proprietate de tipul clasei Y, si invers; un exemplu pentru un One-to-One va fi acesta:
Avem 2 entitati: Student si StudentAddress (adresa studentului respectiv).

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }

    public virtual StudentAddress Address { get; set; }
}

public class StudentAddress
{
    public string City { get; set; }
    public int Zipcode { get; set; }
    public string Country { get; set; }
    public int StudentId { get; set; }

    public virtual Student Student { get; set; }
}
```

One-To-Many

Singura diferenta fata de 1:1 este ca una din clase va avea o lista de obiecte de tipul celeilalte clase; o sa schimbam putin clasele din exemplu:

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }

    public virtual ICollection<StudentGrade> Grades { get; set; }
}

public class StudentGrade
{
    public int Id { get; set; }
    public string Subject { get; set; }
    public int StudentId { get; set; }

    public virtual Student Student { get; set; }
}
```

One-To-One

Aici nu cred ca voi mai scrie un exemplu; este fix rezolvarea din SQL de la baze de date: luam relatia M:M si o impartim in 2 relatii de tipul 1:M.

Hai totusi un exemplu, fara cod: User si Roles... un user poate avea mai multe roluri, iar mai multi useri pot avea acelasi rol; Rezolvarea este adaugarea clasei UserRoles (UserId, RoleId), iar User si Role vor avea un ICollection<UserRole>.

In exemplele de mai sus, datorita denumirilor, EF-ul va sti sa identifice relatiile si detaliile despre fiecare; aditional, pentru a specifica chestii precum un ForeignKey mai explicit (ex: in loc de AddressId s-ar chema "NewAddress" – EF-ul nu va identifica acela ca FK, va trebui sa-I specificam noi), putem folosi **DataAnnotations**. Sunt "notite" pe care le punem deasupra unei proprietati pentru a specifica restrictii asupra lor:

```
[MaxLength(60), Required] // <- Data Annotations
public string Title { get; set; }
```

Acestea sunt, insa limitate: `Key, Required, MaxLength, MinLength, ForeignKey, DatabaseGenerated, NotMapped`. Pe un proiectel simplu, merge folosit doar DataAnnotations, dar pe ceva mai complex, vom avea nevoie de putin mai multa customizare.

Ceva repede despre FluentAPI

Este o configurare mai avansata a traducerii codului dintre C# si baza de date. Pentru a folosi, in Context-ul nostru, va trebui sa-i facem override metodei *OnModelCreating*. Astfel, ar arata cam asa:

```
public class SchoolContext: DbContext
{
    public SchoolDBContext(): base()
    {
    }

    public DbSet<Student> Students { get; set; }
    public DbSet<Standard> Standards { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        //Map entity to table
        modelBuilder.Entity<Student>().ToTable("StudentInfo");
        modelBuilder.Entity<Standard>().ToTable("StandardInfo","dbo");
    }
}
```

Configurarea relatiilor prin FluentAPI ar arata asa:

1:1

```
modelBuilder.Entity<StudentAddress>(entity =>
{
	entity.HasOne(x => x.Student)
		.WithOne(x => x.Address)
		.HasForeignKey<StudentAddress>(x => x.StudentId);
});
```

1:M

```
modelBuilder.Entity<StudentGrade>(entity =>
{
	entity.HasOne(x => x.Student)
		.WithMany(x => x.Grades)
		.HasForeignKey(x => x.StudentId);
});
```

M:M

Room (Id, Number)
Client (Id, CNP, FirstName, LastName, DateRegistered)
Booking (Id, RoomId, PersonId, StartDate, EndDate, IsCancelled)

```
modelBuilder.Entity<Booking>(entity =>
{
	entity.HasKey(x => x.Id);

	entity.HasOne(x => x.Client)
		.WithMany(x => x.Booking)
		.HasForeignKey(x => x.ClientId);

	entity.HasOne(x => x.Room)
		.HasMany(x => x.Booking)
		.HasForeignKey(x => x.RoomId);
});
```