

# Dezvoltarea Aplicațiilor Web utilizând ASP.NET Core MVC

## Curs 5

---

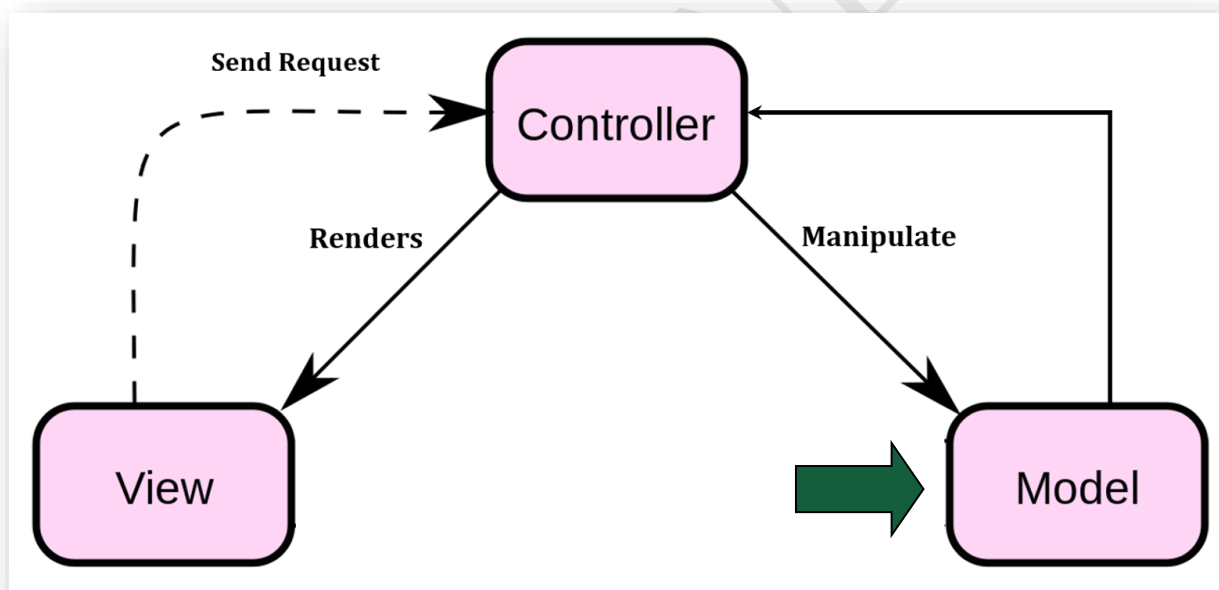
### Cuprins

Model (Stratul business – prelucrarea datelor) .....	2
Ce este Modelul.....	2
Entity Framework Core.....	3
Entity Framework.....	3
Entity Framework Core .....	4
Instalare Entity Framework Core .....	4
Entity Framework Core – Migrații .....	9
Ce sunt migrațiile? .....	9
Crearea unui proiect utilizând EF și sistemul de migrații.....	10
Crearea proiectului .....	10
Adăugare Entity Framework Core .....	10
Adăugarea Modelului.....	10
Conexiunea cu Baza de Date.....	13
Ce este Dependency Injection .....	13
Varianta 1 – fără Dependency Injection.....	16
Varianta 2 – cu Dependency Injection .....	17
Adăugarea unei baze de date SQL Server .....	19
Crearea migrațiilor în baza de date .....	25
C.R.U.D. utilizând Entity Framework.....	27
Index .....	27
Show .....	28
New.....	29
Model Binding.....	30
Edit.....	31
Delete.....	32

## Model (Stratul business – prelucrarea datelor)

### Ce este Modelul

**Modelul** este responsabil cu gestionarea datelor din aplicație și manipularea acestora. Acesta răspunde cererilor care vin din View prin intermediul Controller-ului, modelul comunicând doar cu Controller-ul. Este cel mai de jos nivel, care se ocupă cu **procesarea** și **manipularea** datelor, reprezentând nucleul aplicației, fiind cel care realizează legătura cu baza de date. **Modelul** oferă accesul la date prin intermediul atributelor publice ale claselor.



# Entity Framework Core

## Entity Framework

Pentru stocarea datelor în ASP.NET MVC, se utilizează o tehnologie open source numită **Entity Framework (EF)**.

Entity Framework este un **ORM (Object Relational Mapper)** pentru .NET, și anume este o colecție de librării care corelează fiecare clasă dintr-un model cu o bază de date. Scopul utilizării EF este acela de a permite dezvoltatorilor să se concentreze pe dezvoltarea propriu-zisă a aplicației și nu pe baza de date.

Procesarea datelor se poate realiza și prin metode clasice, de exemplu utilizând ADO.NET, dar EF oferă posibilitatea implementării eficiente a operațiilor de tip CRUD (Create, Read, Update, Delete).

De asemenea, în cadrul EF se poate utiliza **LINQ (Language Integrated Query)**, ajutând la integrarea oricărui **RDBMS (Relational Database Management System)** → Oracle Database, SQL Server, etc. Un RDBMS este un **sistem de management al bazelor de date relaționale**, care stochează date într-un mod structurat, utilizând tabele, pe care ulterior le accesează și prelucrează cu ajutorul unui limbaj **SQL (Structured Query Language)**. Un RDBMS asigură securitatea, integritatea și consistența datelor.

**LINQ** este o componentă a .NET Framework care permite interogarea direct în limbajul de programare C#, permițând dezvoltatorilor să scrie interogări într-un mod mult mai natural și integrat. LINQ permite efectuarea de interogări pe diverse surse de date, inclusiv array-uri, colecții de obiecte, XML și baze de date relaționale.

## Entity Framework Core

**Entity Framework Core** este versiunea mai light a EF, cross-platform (Linux, Windows) care funcționează foarte bine împreună cu ASP.NET Core. Entity Framework Core suportă, la fel ca EF, atât tehnica **database-first** cât și **code-first**.

EF Core conține atât posibilitatea integrării unui RDBMS (Oracle Database, Microsoft SQL Server, MySQL), cât și integrarea unor baze de date non-relaționale (MongoDB, Redis, CassandraDB).

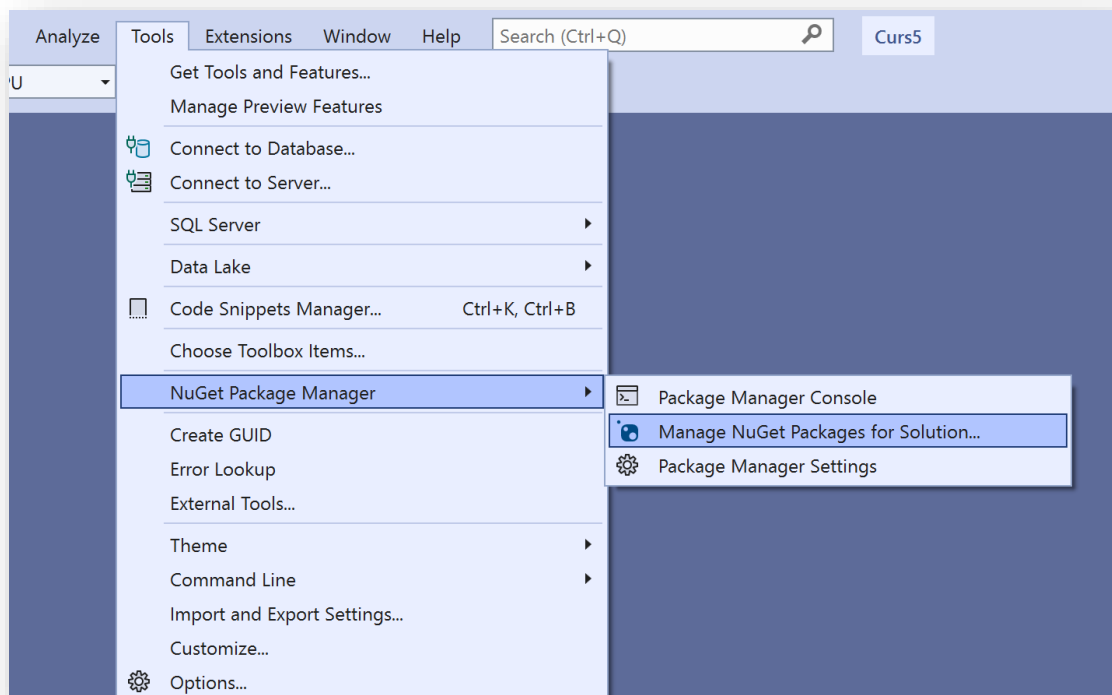
## Instalare Entity Framework Core

Entity Framework este un pachet care poate fi instalat folosind **NuGet** și care suportă tehnica **code-first**. Tehnica code-first oferă posibilitatea dezvoltatorilor de a scrie clase prin intermediul cărora baza de date va fi generată automat. Acest lucru duce la o dezvoltare curată și rapidă a aplicațiilor cu baze de date.

Pentru instalarea Entity Framework (EF) se procedează astfel:

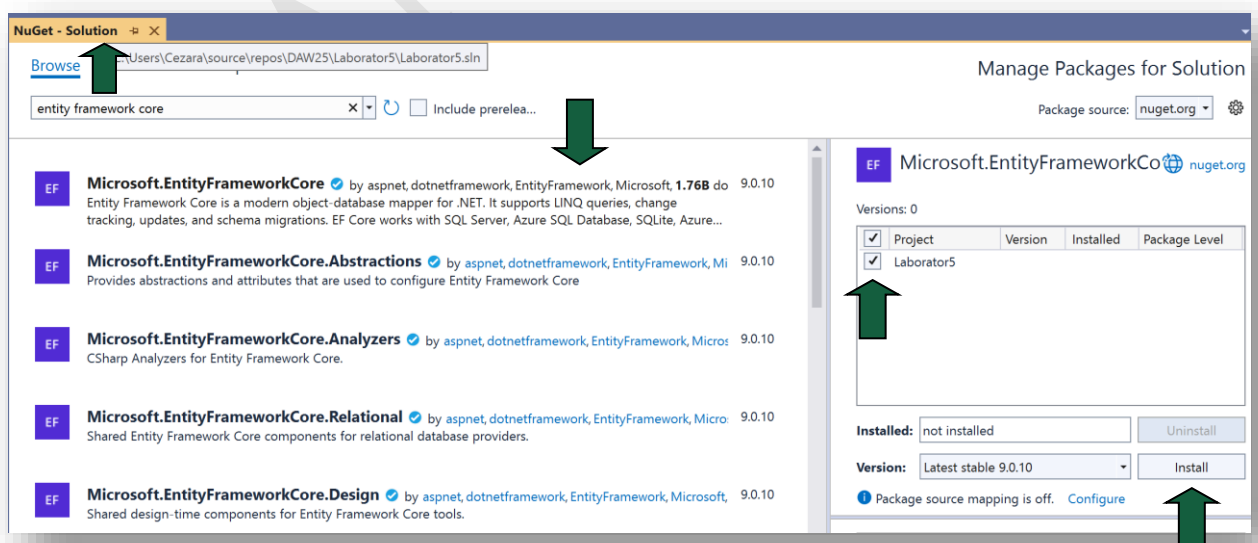
### PASUL 1:

Tools → NuGet Package Manager → Manage NuGet Packages for Solution...

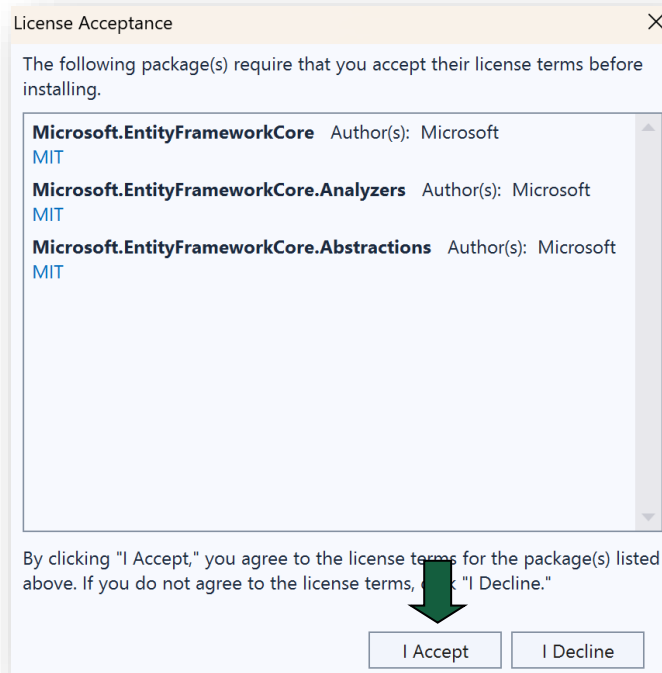


## PASUL 2:

Se accesează opțiunea **Browse** și se caută **Microsoft.EntityFrameworkCore**, după cum urmează:

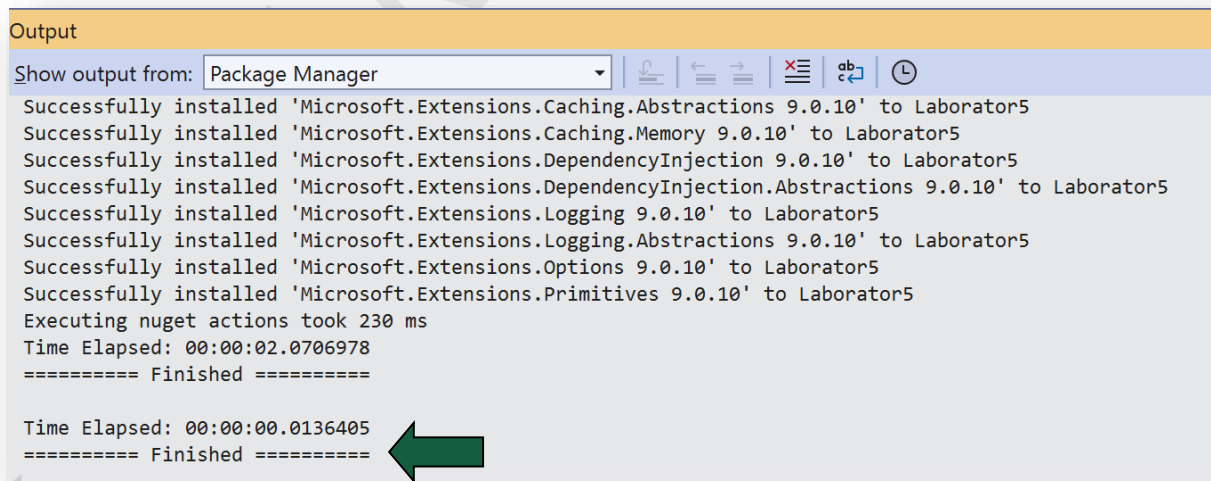


### PASUL 3:

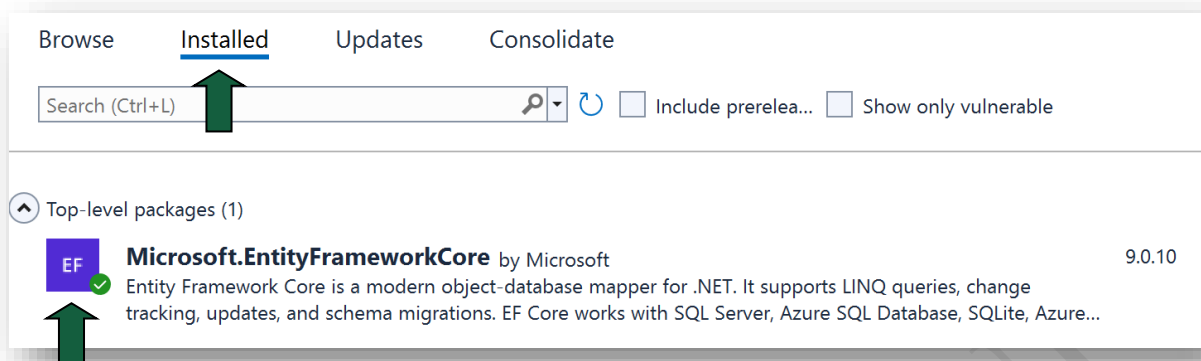


### PASUL 4:

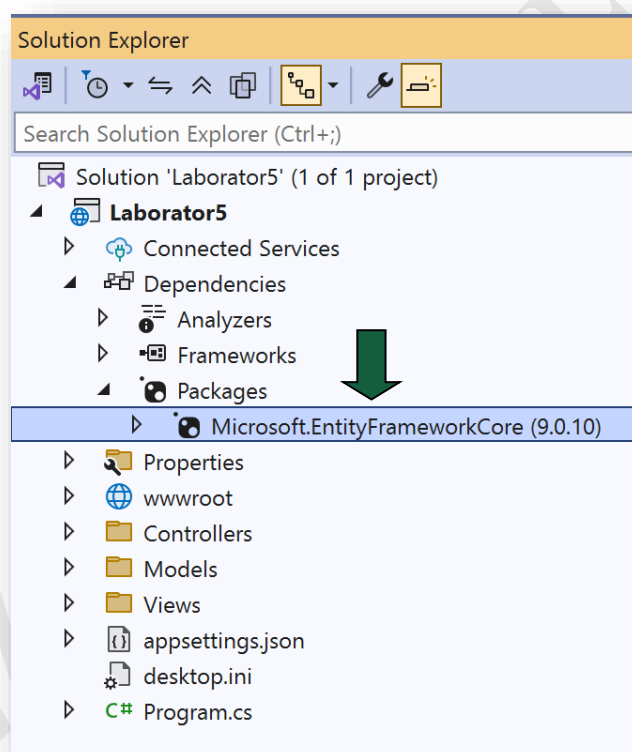
Dupa instalare se poate verifica în consolă dacă EF a fost adăugat cu succes în cadrul proiectului.



De asemenea, se poate verifica și în secțiunea **Installed**.

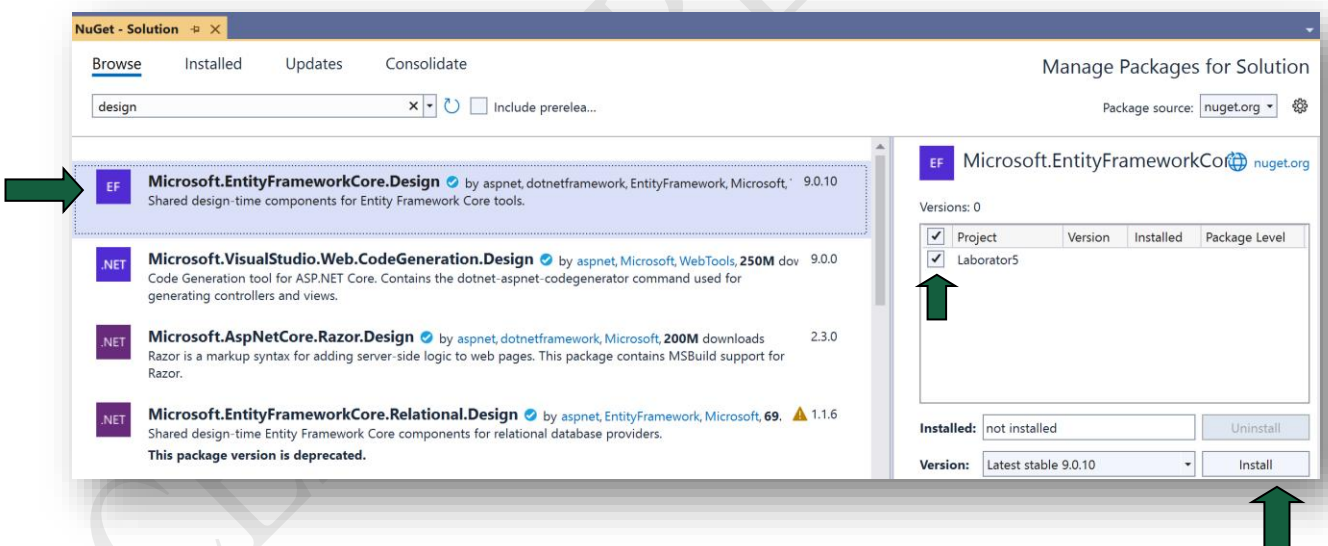
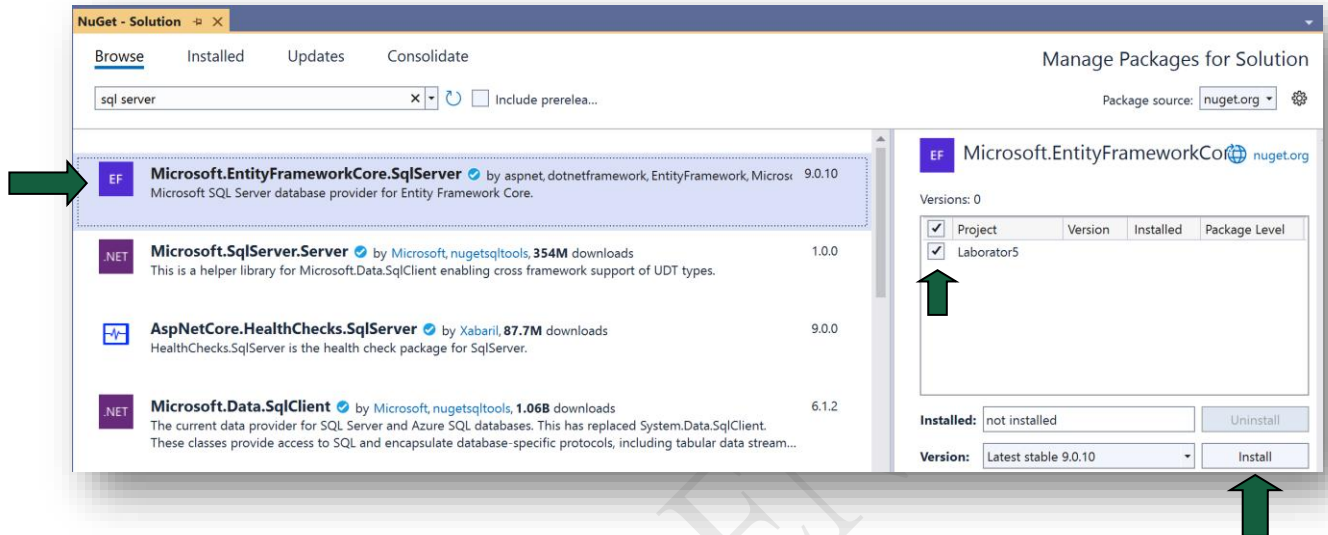


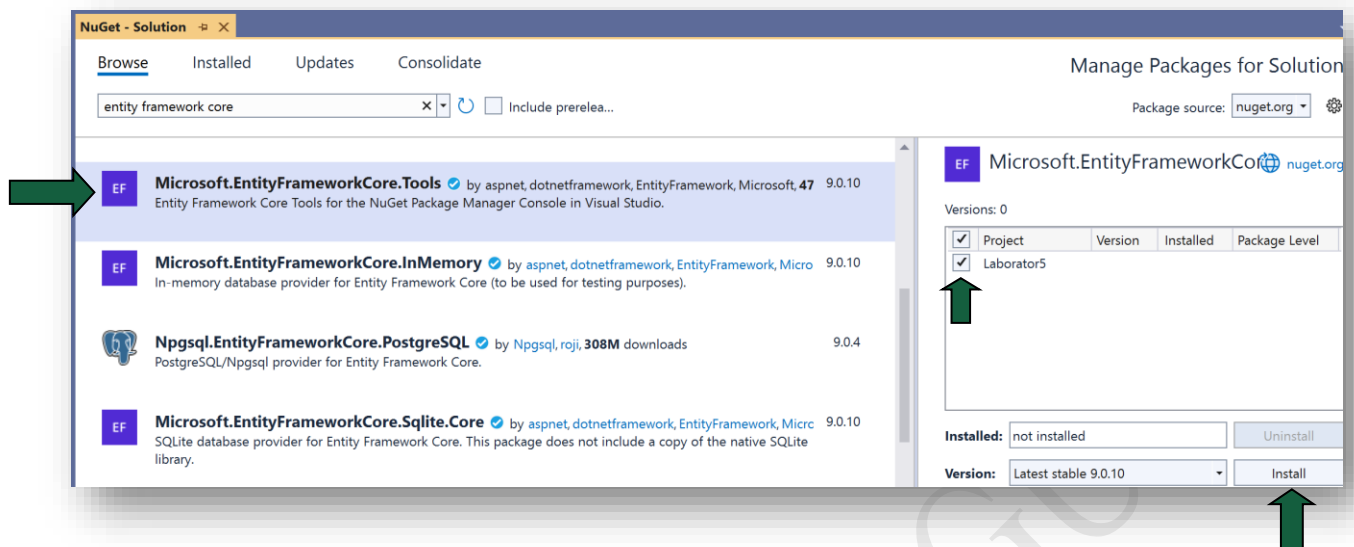
Sau chiar în Solution Explorer → Dependencies → Packages



În continuare este necesară includerea pachetelor **SqlServer** (pentru baza de date), **Design** (pachet care conține tool-urile necesare pentru rularea comenzilor de migrare) și pachetul **Tools**, care include comenzi precum: Add-Migration, Drop-Database, Get-DbContext, Remove-Migration, etc.

- ➔ Microsoft.EntityFrameworkCore.SqlServer
- ➔ Microsoft.EntityFrameworkCore.Design
- ➔ Microsoft.EntityFrameworkCore.Tools





## Entity Framework Core – Migrații

### Ce sunt migrațiile?

În timpul dezvoltării unei aplicații, baza de date se modifică constant, fiind necesare entități noi, proprietăți noi sau chiar eliminarea unor proprietăți existente. Pentru a sincroniza aceste modificări cu baza de date existentă, sunt necesare **migrații**.

În momentul în care apare o modificare în baza de date, Entity Framework Core, prin sistemul de migrații, compară modelul curent cu cel anterior pentru a detecta diferențele dintre cele două versiuni. Ulterior este generat un fișier, conținând codul asociat migrației.

# Crearea unui proiect utilizând EF și sistemul de migrații

## Crearea proiectului

Se creează un nou proiect, procedând la fel ca în cursurile anterioare. Proiectul o să se numească **Curs5**.

## Adăugare Entity Framework Core

În cadrul noului proiect se adaugă EF (**VEZI** Secțiunea – Instalare Entity Framework Core – din cadrul cursului curent).

## Adăugarea Modelului

Pentru adăugarea unui model se pornește de la crearea în cadrul acestuia a tuturor entităților. Prin **entitate** ne referim la un tabel din baza de date.

**Entitate** = un loc, o acțiune, o persoană, etc.

**Exemplu de entități** dintr-o baza de date care gestionează o Universitate

→ Studenți, Cursuri, Note, MergeLa – tabel asociativ între Studenți și Cursuri

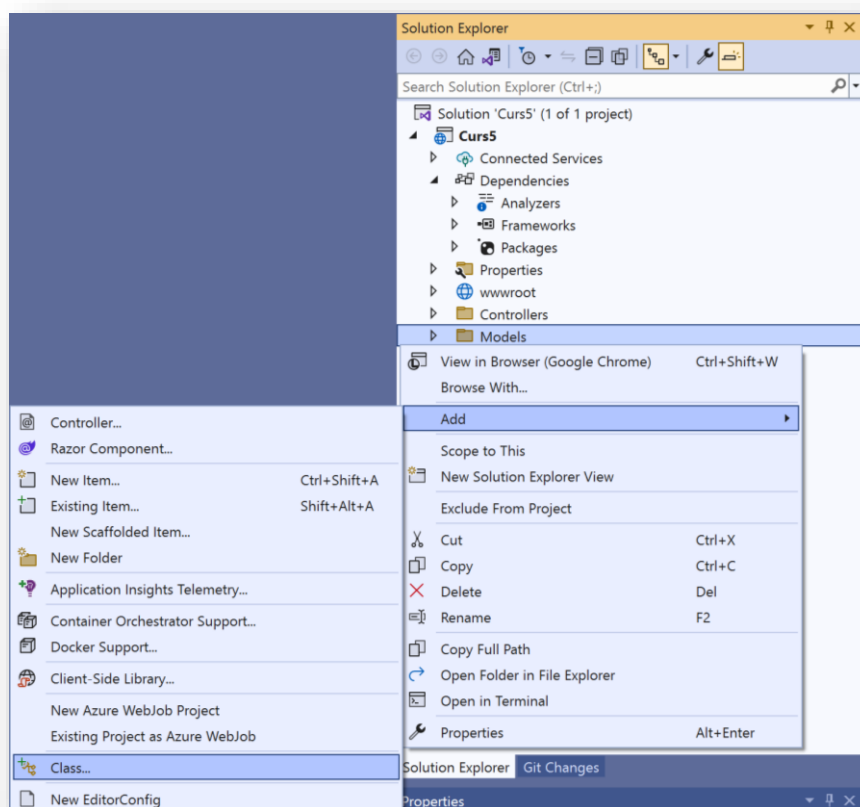
În continuare vom adăuga clasa **Student** cu următoarele atribute:

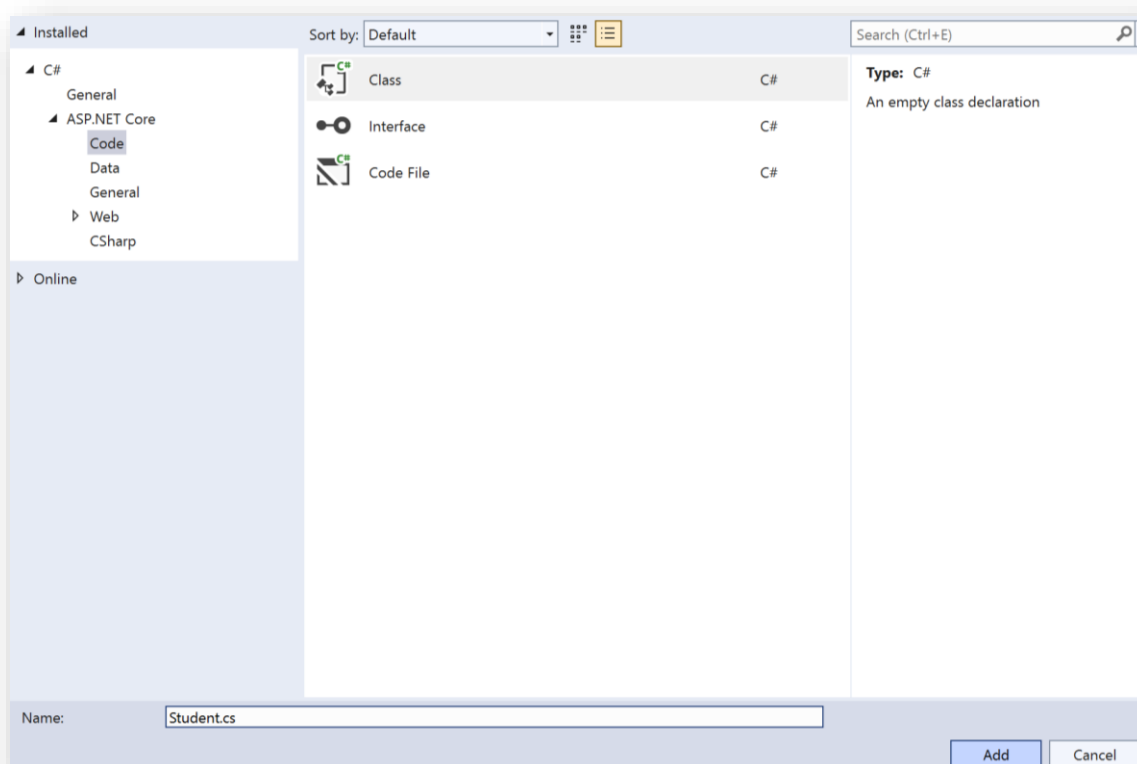
- **StudentId** – de tip int care reprezintă ID-ul studentului;
- **Name** – de tip string care reprezintă numele studentului;
- **Email** – de tip string care reprezintă adresa de e-mail a studentului;
- **CNP** – de tip string care reprezintă CNP-ul studentului. Această proprietate a fost definită de tip string deoarece spațiul alocat pentru int nu suportă valori de 13 caractere. De asemenea, definit

ca string se poate procesa caracter cu caracter pentru calcule ulterioare (ex: extragere dată de naștere).

Pentru adăugarea clasei **Student** se parcurg următorii pași:

Click dreapta Model → Add → Class → ASP.NET Core → Class → se completează numele clasei **Student.cs**





Clasa **Student** din fișierul **Student.cs**:

```
public class Student
{
    public int StudentId { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public string CNP { get; set; }
}
```

Această clasă reprezintă tabelul **Student** din baza de date, pe care în varianta clasică îl adăugăm prin intermediul comenzii **CREATE TABLE**.

Atributele / proprietățile clasei sunt: StudentId, Name, Email, CNP. Acestea reprezintă coloanele din tabelul Student, coloane care sunt create în momentul în care se creează și tabelul → utilizând comanda **CREATE TABLE**

## Conexiunea cu Baza de Date

Conexiunea cu baza de date se poate realiza în două moduri:

- fără dependency injection
- cu dependency injection

## Ce este Dependency Injection

**Dependency Injection (DI)** este un design pattern folosit în programare pentru a abstractiza implementarea și a asigura cuplarea rapidă între componentele software. În esență, acest pattern permite injectarea dependențelor (**de exemplu**: obiecte ale claselor, servicii sau resurse) într-o componentă, în loc ca acea componentă să-și creeze singură dependențele. Astfel, codul devine modular și mult mai ușor de testat.

Framework-ul ASP.NET Core are un sistem de **Dependency Injection** integrat, care este configurat și utilizat prin intermediul clasei **Program.cs**. Acesta permite dezvoltatorilor să înregistreze dependențele într-un container de servicii, iar aceste servicii pot fi apoi injectate (inserate) automat, la runtime, acolo unde sunt necesare, prin constructori, metode sau proprietăți. Astfel, un Controller poate cere o instanță a unui obiect din cadrul framework-ului doar prin specificarea unui parametru și tipul acestuia.

Un astfel de exemplu este conexiunea la baza de date, care se cere prin intermediul constructorului, specificându-se un parametru de tip `AppDbContext` (în cazul nostru particular din curs unde am numit clasa în acest mod). În acest caz, framework-ul știe în mod automat să returneze o instanță a `AppDbContext` din serviciul aferent bazei de date, configurat în `Program.cs`.

**Dependency Injection** este o parte fundamentală a arhitecturii moderne a aplicațiilor web în ASP.NET Core, îmbunătățind modul în care componentele software interacționează și sunt testate.

### Exemplu din viața reală pentru o mai bună înțelegere a conceptului:

Să ne imaginăm că bucătăria unui restaurant este o aplicație software, iar bucătarul este o componentă din această aplicație. În această bucătărie sunt diferite unelte și ingrediente necesare pentru a prepara diverse feluri de mâncare.

#### Fără Dependency Injection:

Fără DI, fiecare bucătar (componentă) trebuie să își aducă propriile unelte și ingrediente sau să le creeze de la zero de fiecare dată când începe să gătească. Acest lucru poate fi ineficient și consumator de timp, deoarece fiecare bucătar trebuie să știe exact unde sunt toate ingredientele și uneltele, să le transporte cu el și să le configureze.

#### Cu Dependency Injection:

DI în contextul exemplului curent funcționează ca un manager sau un asistent care pregătește și furnizează toate uneltele și ingredientele necesare fiecărui bucătar atunci când are nevoie de acestea. Bucătarul nu trebuie să cunoască sursa ingredientelor sau unde sunt stocate uneltele. El pur și simplu le primește gata de utilizat și se poate concentra pe ceea ce face cel mai bine – gătitul.

#### 1. Injectarea ingredientelor (Dependențelor):

- În ASP.NET Core, containerul DI este ca un manager de bucătărie care știe ce ingrediente și unelte sunt necesare fiecărui bucătar și le pregătește în avans;

## 2. Configurarea inițială (Serviciile de Înregistrare):

- La fel cum un manager de restaurant ar putea decide ce ingrediente și echipamente sunt necesare în bucătărie și le-ar pregăti înainte ca bucătarii să vină, în ASP.NET Core, dependențele (cum ar fi serviciile de bază de date, logistica sau API-uri externe) sunt configurate și înregistrate în containerul de DI în clasa Program;

## 3. Utilizarea dependențelor:

- Când vine timpul să gătească, bucătarul pur și simplu solicită managerului ingredientele și uneltele necesare. În ASP.NET Core, când o componentă (**de exemplu:** un Controller) are nevoie de un serviciu (cum ar fi accesul la baza de date), acesta este automat furnizat de containerul de DI;

Se observă cum DI ajută la separarea sarcinilor într-o aplicație, permițând fiecărei componente să se concentreze pe sarcinile sale specifice, fără a se îngrijora de detaliile legate de obținerea resurselor necesare. Prin urmare, DI **îmbunătățește organizarea, eficiența și flexibilitatea** aplicațiilor software.

## Varianta 1 – fără Dependency Injection

Pentru a putea adăuga layer-ul de conexiune cu baza de date în cadrul unui model, este necesară adăugarea unei noi clase.

Se adaugă o clasă, numită sugestiv **AppDbContext**.

```
public class AppDbContext : DbContext
{
    public AppDbContext() : base ()
    {
    }

    protected override void OnConfiguring(
        DbContextOptionsBuilder options)
    {
        options.UseSqlServer(
            @"Stringul de Conexiune");
    }

    public DbSet<Student> Students { get; set; }
}
```

Clasa **AppDbContext** moștenește clasa de bază **DbContext** din Entity Framework. Clasa de bază realizează în mod automat conexiunea cu baza de date, crearea tabelului dacă acesta nu există și conține o proprietate **DbSet**, care trebuie să primească tipul modelului (Student în cazul curent) și numele pluralizat al modelului.

**DbSet <Student> Students { get; set; }** – prin intermediul acestei secvențe de cod vom avea acces la intrările din baza de date; se pot **interoga** și **stoca** instanțe de tip Student.

Stringul de conexiune la baza de date este preluat de parametrul **options** din metoda **OnConfiguring**, prin intermediul următoarei secvențe de cod:

```
options.UseSqlServer(@"Stringul de Conexiune");
```

**UseSqlServer()** → metodă prin intermediul căreia se configurează contextul pentru conectarea la baza de date. Primește ca argument stringul de conectare la baza de date. **Stringul de conectare la baza de date se obține urmând secțiunea următoare din curs.**

În final se adaugă în **Program.cs** următoarea secvență de cod pentru inițializarea bazei de date.

```
builder.Services.AddDbContext<AppDbContext>();
```

În cadrul fiecărui Controller trebuie să se instanțieze contextul pentru realizarea conexiunii la baza de date.

```
private AppDbContext db = new AppDbContext();
```

## Varianța 2 – cu Dependency Injection

Se adaugă aceeași clasă, numită sugestiv **AppDbContext** care în acest caz o să aibă doar constructorul, astfel:

```
public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext>
options)
        : base(options)
    {
    }
    public DbSet<Student> Students { get; set; }
}
```

Conexiunea cu baza de date se va realiza în acest caz exclusiv în **Program.cs**.

```
var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");

builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer(connectionString));
```

În acest caz, stringul de conexiune o să fie în **Solution Explorer** → **appsetting.json** adăugând secvența de cod marcată cu BOLD și acoladă.

```
{
  "ConnectionStrings": {
    "DefaultConnection":
    "Server=(localdb)\\mssqllocaldb;Database=aspnet-53bc9b9d-9d6a-
    45d4-8429-
    2a2761773502;Trusted_Connection=True;MultipleActiveResultSets=tr
    ue"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

În final, în cadrul fiecărui Controller, se realizează conexiunea cu baza de date astfel:

```
public class ArticlesController : Controller
{
    private readonly AppDbContext db;

    public ArticlesController(AppDbContext context)
    {
        db = context;
    }

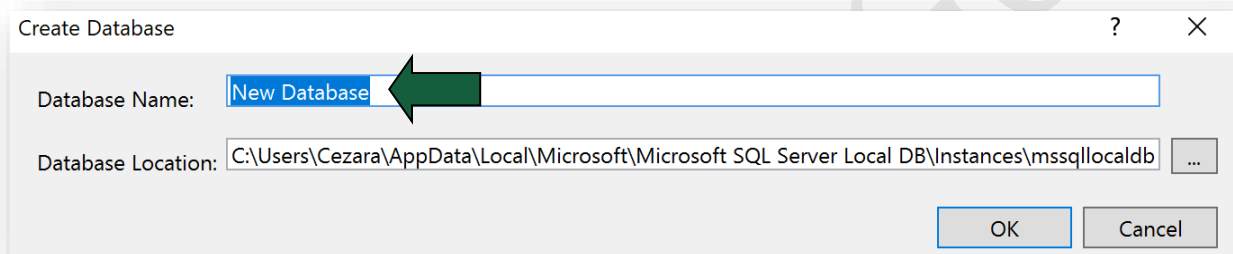
    ...}
}
```

## Adăugarea unei baze de date SQL Server

Pentru adăugarea bazei de date se parcurg următorii pași:

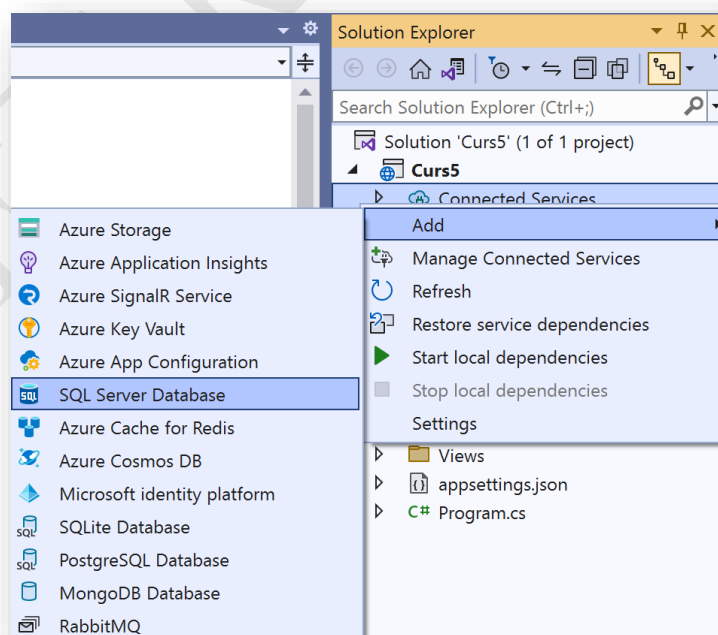
### PASUL 1:

În SQL Server Object Explorer (se află în meniul View) → se deschide secțiunea (localdb)\MSSQLLOCALDB → în folderul Databases → click dreapta → Add new database → se completează numele bazei de date (se alege un nume pe care o să îl folosim în cadrul următorilor pași) → OK



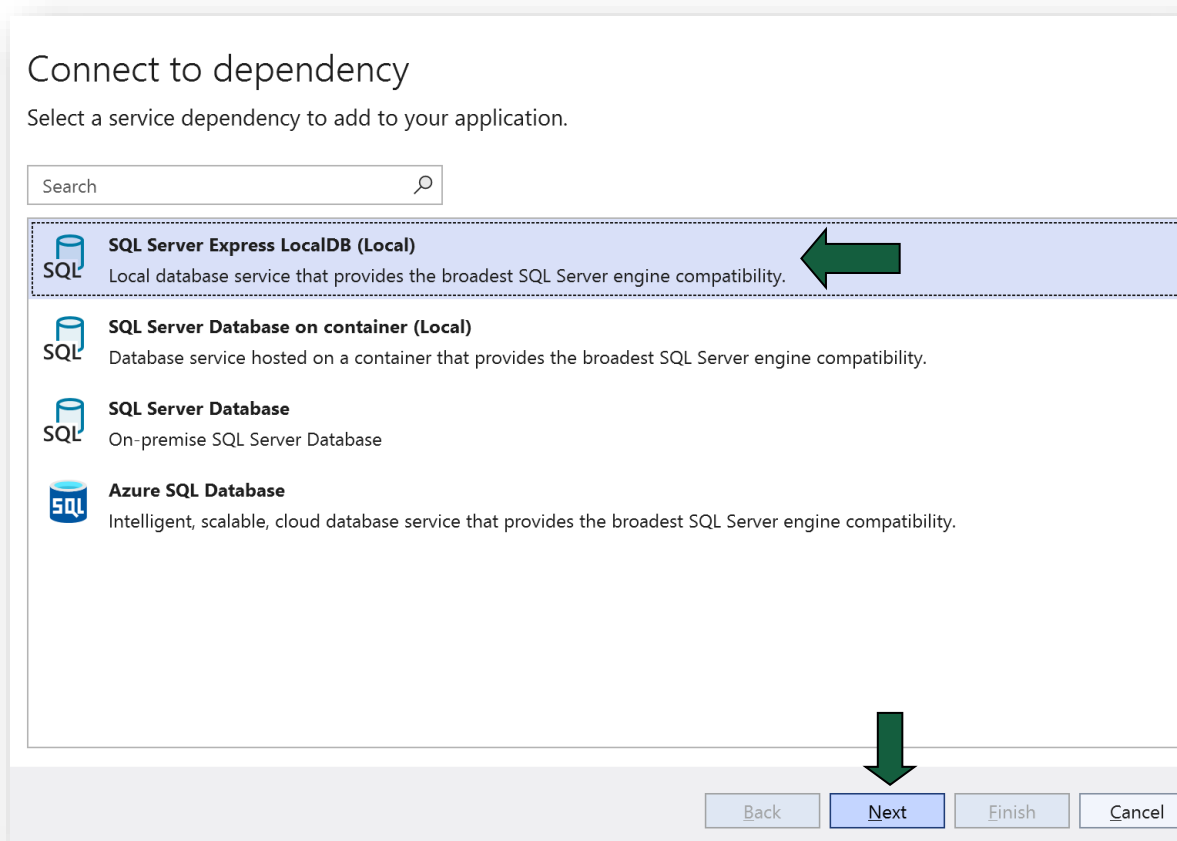
### PASUL 2:

În Solution Explorer → click dreapta pe secțiunea Connected Services → Add → SQL Server Database



### PASUL 3:

Se selectează opțiunea SQL Server Express LocalDB (Local)



#### PASUL 4:

Se adaugă un nume conexiunii la baza de date (ex: StudentsDB, ArticlesDB, etc).

Se bifează opțiunea **None**.

Se modifică string-ul de conexiune apăsând cele 3 puncte (unde se află cercul roșu), din dreptul casuței Connection String Value.

Se apasă butonul Next.

Connect to SQL Server Express LocalDB (Local)

Provide connection string and specify how to save it

Connection string name  
ConnectionStrings:StudentsDB

Connection string value  
Server=(localdb)\mssqllocaldb;Database=aspnet-53bc9b9d-9d6a-45d4-8429-2a2761773502;Trusted\_Connection=True;MultipleActive...

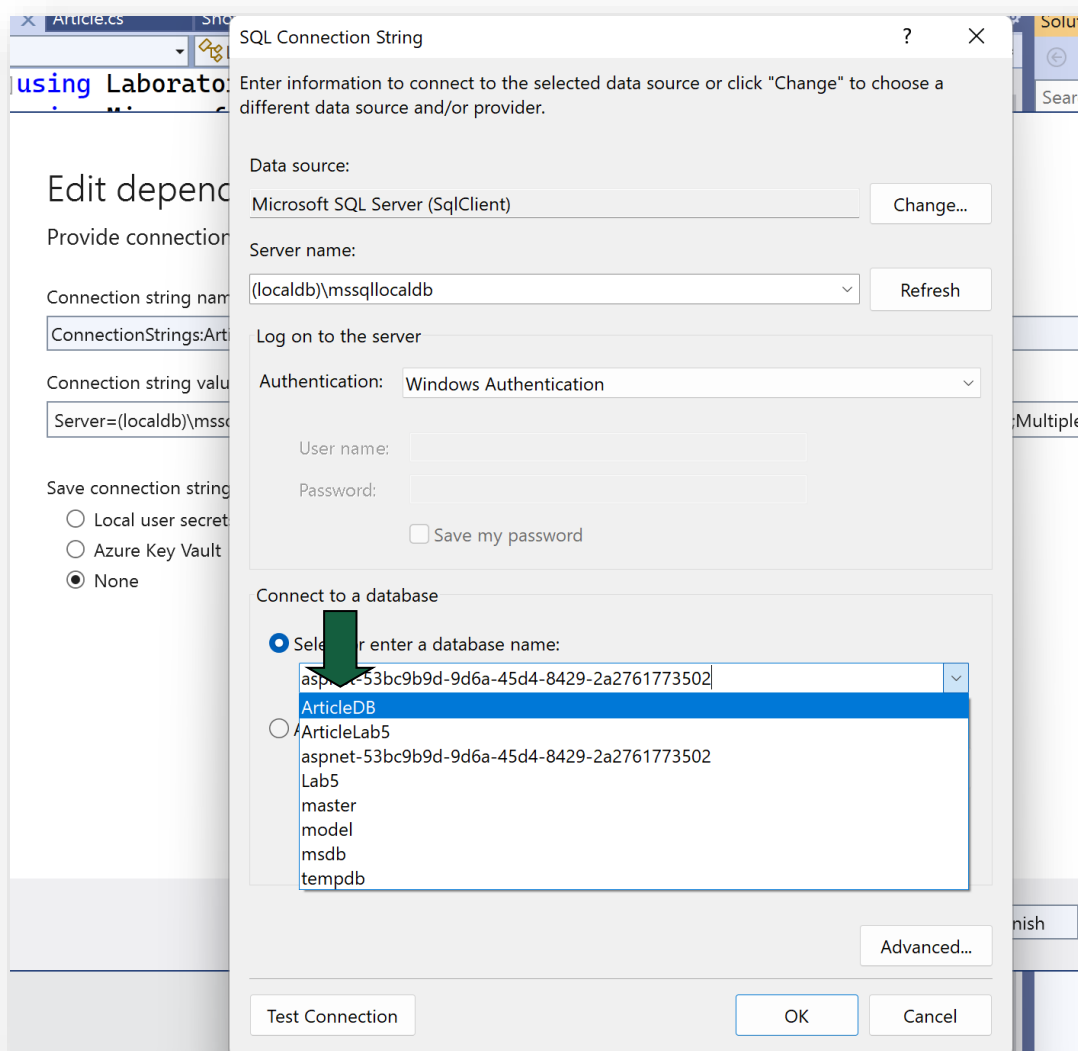
Save connection string value in [Learn more](#)

☐ Local user secrets file  
☐ Azure Key Vault  
☒ None

Back Next Finish Cancel

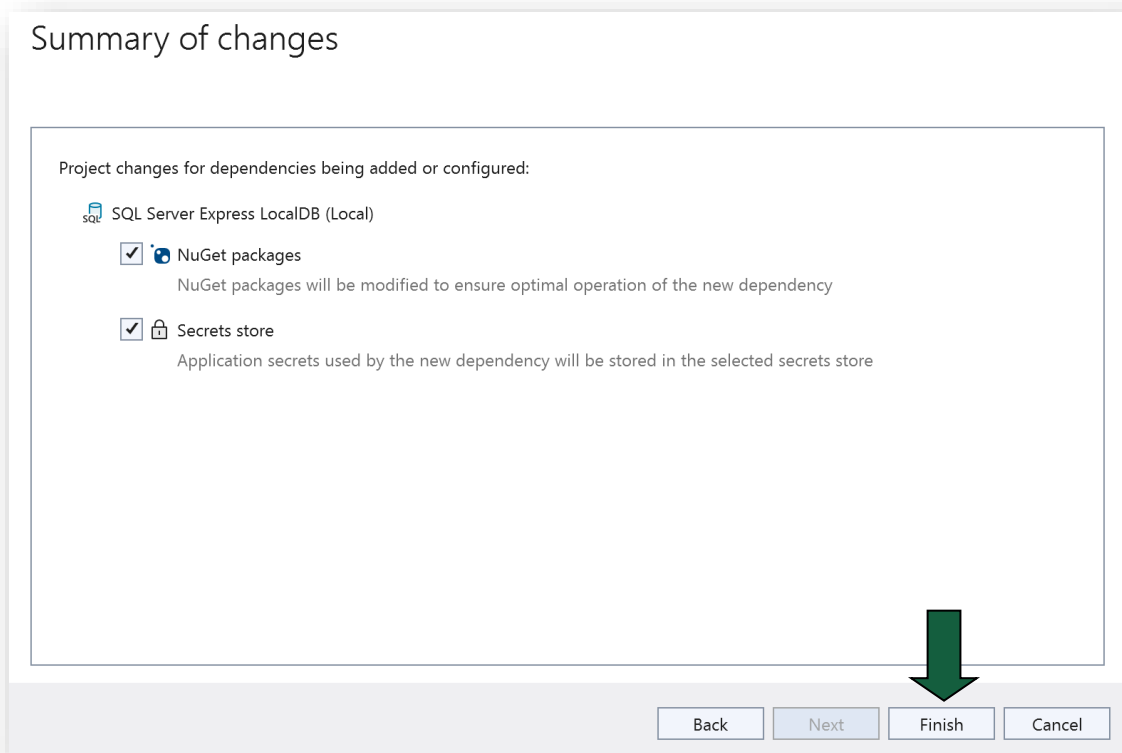
## PASUL 5:

Se selectează baza de date creată la Pasul 1.



Se apasă OK → după care se copiază stringul de conexiune (se utilizează la PASUL 7).

## PASUL 6:



## PASUL 7:

Se adaugă stringul de conectare la baza de date ca parametru al metodei `UseSqlServer()` în `OnConfiguring` (valabil pentru metoda – fără DI).

```
options.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=aspnet-53bc9b9d-9d6a-45d4-8429-2a2761773502;Trusted_Connection=True;MultipleActiveResultSets=true");
```

Dacă se utilizează varianta 2 – cea cu **Dependency Injection** – se adaugă stringul de conexiune în **Solution Explorer** → **appsetting.json** adăugând secvența de cod marcată cu BOLD și acoladă.

```
{
  "ConnectionStrings": {
    "DefaultConnection":
    "Server=(localdb)\mssqllocaldb;Database=aspnet-53bc9b9d-9d6a-45d4-8429-2a2761773502;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

### **!/ OBSERVAȚIE**

Stringul dat ca parametru în metoda de mai sus, `UseSqlServer`, este unic în cadrul fiecărei baze de date. Așadar, nu trebuie să utilizați stringul din acest exemplu, ci trebuie să preluați stringul de conexiune la baza voastră de date, urmând pașii anteriori.

În acest moment proiectul conține Entity Framework și are creată o bază de date și o conexiune cu aceasta. **Urmează să integrăm sistemul de migrații.**

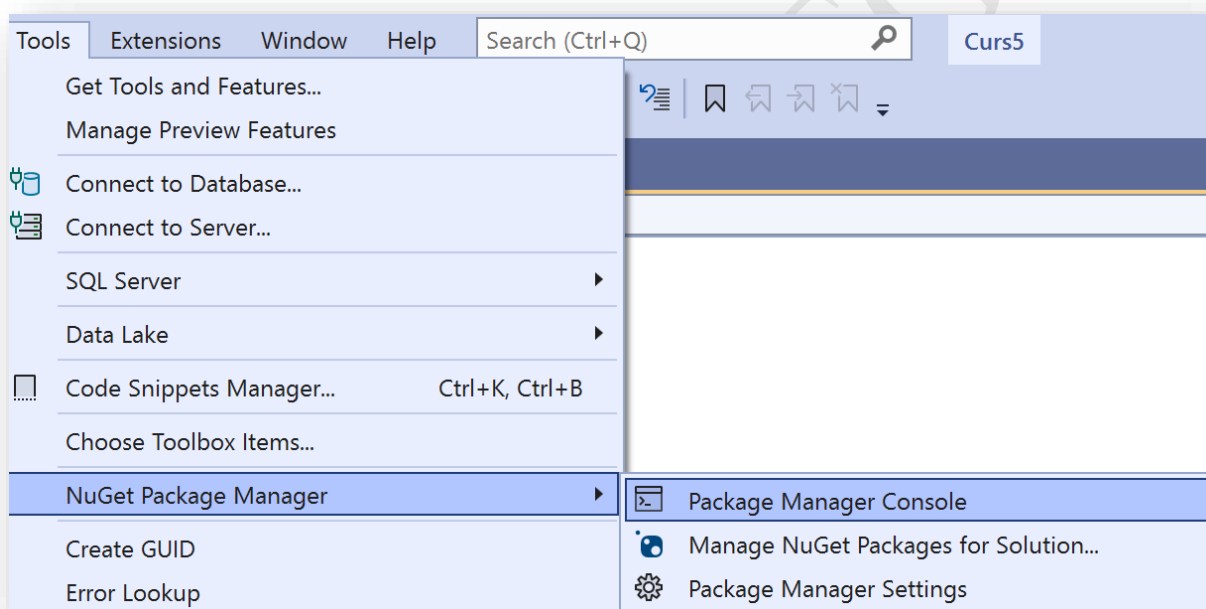
## Crearea migrațiilor în baza de date

Pentru integrarea sistemului de migrații se parcurg pașii următori:

### PASUL 1:

Pentru adăugarea migrațiilor o să se utilizeze consola.

**Tools → NuGet Package Manager → Package Manager Console**



### PASUL 2:


Se adaugă migrația utilizând comanda **Add-Migration** urmată de o denumire pe care o dăm acestei migrații.

```
Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages. Some packages may include dependencies which are governed by additional licenses. Follow the package source (feed) URL to determine any dependencies.

Package Manager Console Host Version 6.3.0.131

Type 'get-help NuGet' to see all available NuGet commands.

PM> Add-Migration CreateStudent
```




După executarea migrației, în Solution Explorer o să se creeze un folder numit **Migrations**, unde o să se genereze fișierele specifice.

### PASUL 3:

Se execută comanda **Update-Database** care modifică baza de date, aducând-o la versiunea finală.

```
Package source: All | Default project: Curs5

Microsoft.EntityFrameworkCore.Infrastructure[10403]
    Entity Framework Core 6.0.10 initialized 'AppDbContext' using provider
    'Microsoft.EntityFrameworkCore.SqlServer:6.0.10' with options: None
    To undo this action, use Remove-Migration.
    PM> Update-Database
```



## C.R.U.D. utilizând Entity Framework

În următoarea parte a cursului vom implementa operațiile de tip CRUD asupra entității Student, utilizând Entity Framework.

### Index

```
private readonly ApplicationDbContext db;

public ArticlesController(ApplicationDbContext context)
{
    db = context;
}

public IActionResult Index()
{
    var students = from student in db.Students
                   orderby student.Name
                   select student;

    ViewBag.Students = students;

    return View();
}
```

Preluăm toți studenții din baza de date, ordonați după nume, prin intermediul db.Students

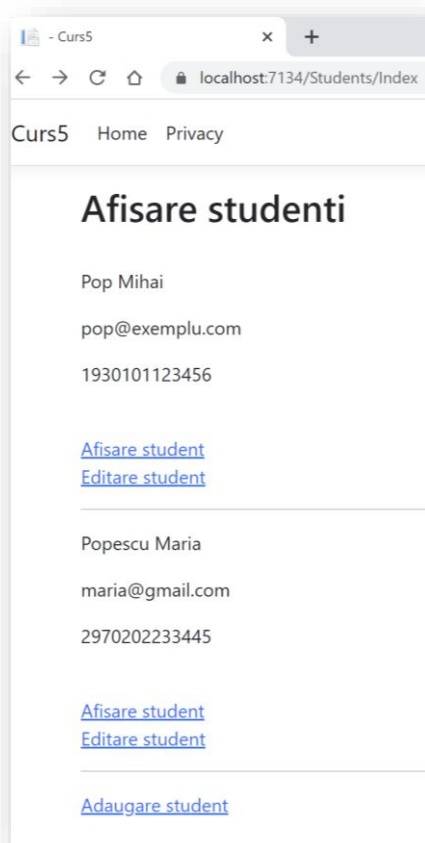
### Index.cshtml

```
<h2>Afisare studenti</h2>
<br />

@foreach (var student in ViewBag.Students)
{
    <p>@student.Name</p>
    <p>@student.Email</p>
    <p>@student.CNP</p>

    <br />
    <a href="/Students/Show/@student.StudentId">Afisare
student</a>
    <br />
    <a href="/Students/Edit/@student.StudentId">Editare
student</a>
    <hr />
}

<a href="/Students/New">Adaugare student</a>
```



## Show

```
public ActionResult Show(int id)
{
    Student student = db.Students.Find(id);
    ViewBag.Student = student;
    return View();
}
```

Metoda Find() primește ca parametru o valoare pentru coloana care este cheie primară.  
-id este parametrul din rută

## Show.cshtml

```
<h2>Afisare student</h2>

<br />

<p>@ViewBag.Student.Name</p>
<p>@ViewBag.Student.Email</p>
<p>@ViewBag.Student.CNP</p>

<br />
<a href="/Students/Index">Afisare studenti</a>
```

## New

```
public IActionResult New()
{
    return View();
}

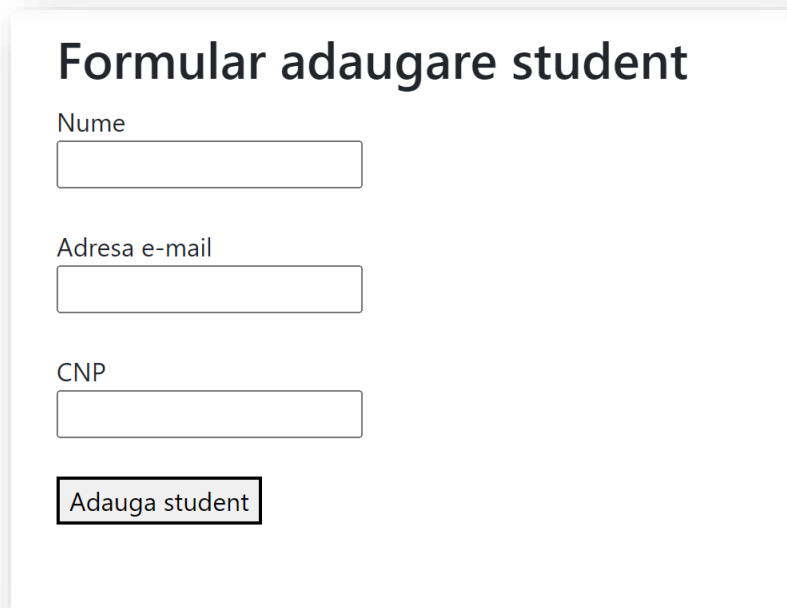
[HttpPost]
public IActionResult New(Student s)
{
    try
    {
        db.Students.Add(s);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    catch (Exception)
    {
        return View();
    }
}
```

**Students.Add** primește ca parametru un obiect de tip **Student**, iar **SaveChanges** va face commit în baza de date

## New.cshtml

```
<h2>Formular adaugare student</h2>

<form method="post" action="/Students/New">
    <label>Nume</label>
    <br />
    <input type="text" name="Name" />
    <br /><br />
    <label>Adresa e-mail</label>
    <br />
    <input type="text" name="Email" />
    <br /><br />
    <label>CNP</label>
    <br />
    <input type="text" name="CNP" />
    <br />
    <br />
    <button type="submit">Adauga student</button>
</form>
```



**Formular adaugare student**

Nume

Adresa e-mail

CNP

## Model Binding

În ASP.NET MVC, **model binding** ne permite să facem legătura între request-urile de tip HTTP și un Model. Model binding este procesul de creare a obiectelor folosind datele trimise de browser printr-un request HTTP, prin intermediul formularelor din View.

Model binding este o legătură între request-urile HTTP și metodele unui Controller (Acțiuni). Deoarece datele trimise prin POST sau GET ajung întotdeauna la Controller, acest mecanism de binding leagă în mod automat variabilele de request cu attributele publice ale modelului. Această mapare se va face după **numele atributelor modelului**.

```
<label>Nume</label>
```

```
<input type="text" name="Name" />
```

```
<label>Adresa e-mail</label>
```

```
<input type="text" name="Email" />
```

```
<label>CNP</label>
```

```
<input type="text" name="CNP" />
```

Parametrii care se vor trimite prin request la controller

### !/\ OBSERVAȚIE

Este necesar ca numele câmpurilor din View să coincidă cu numele atributelor din Model pentru ca binding-ul să funcționeze.

### Edit

```
public IActionResult Edit(int id)
{
    Student student = db.Students.Find(id);
    ViewBag.Student = student;
    return View();
}

[HttpPost]
public ActionResult Edit(int id, Student requestStudent)
{
    Student student = db.Students.Find(id);

    try
    {
        student.Name = requestStudent.Name;
        student.Email = requestStudent.Email;
        student.CNP = requestStudent.CNP;
        db.SaveChanges();

        return RedirectToAction("Index");
    }
    catch (Exception)
    {
        return RedirectToAction("Edit", student.StudentId);
    }
}
```

## Edit.cshtml

```
<h2>Editare student</h2>

<br />

<form method="post"
action="/Students/Edit/@ViewBag.Student.StudentId">

    <label>Nume</label>
    <br />
    <input type="text" name="Name" value="@ViewBag.Student.Name" />
    <br /><br />
    <label>Adresa e-mail</label>
    <br />
    <input type="text" name="Email" value="@ViewBag.Student.Email" />
    <br /><br />
    <label>CNP</label>
    <br />
    <input type="text" name="CNP" value="@ViewBag.Student.CNP" />
    <br />
    <button type="submit">Modifica student</button>

</form>
```

## Delete

```
[HttpPost]
public ActionResult Delete(int id)
{
    Student student = db.Students.Find(id);
    db.Students.Remove(student);
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

**Remove** primește ca parametru un obiect de tip **Student**. **SaveChanges** salvează modificările

## Show.cshtml (se va utiliza view-ul show)

```
<form method="post"
action="/Students/Delete/@ViewBag.Student.StudentId">

    <button type="submit">Sterge studentul</button>

</form>
```

### **!/ OBSERVAȚIE**

În momentul în care sunt necesare în baza de date, fie adăugări sau ștergeri de tabele, fie adăugări sau ștergeri de coloane sau proprietăți, este nevoie de o nouă migrație în baza de date.

**De exemplu:** dacă se dorește adăugarea atributului **Address** în clasa Student → `public string Address { get; set; }`

Se adaugă proprietatea, după care se execută o nouă migrație:

→ Add-Migration AddAddressToStudent

→ Update-Database