

# Dezvoltarea Aplicatiilor Web utilizand ASP.NET Core MVC

## Curs 1

---

### Cuprins

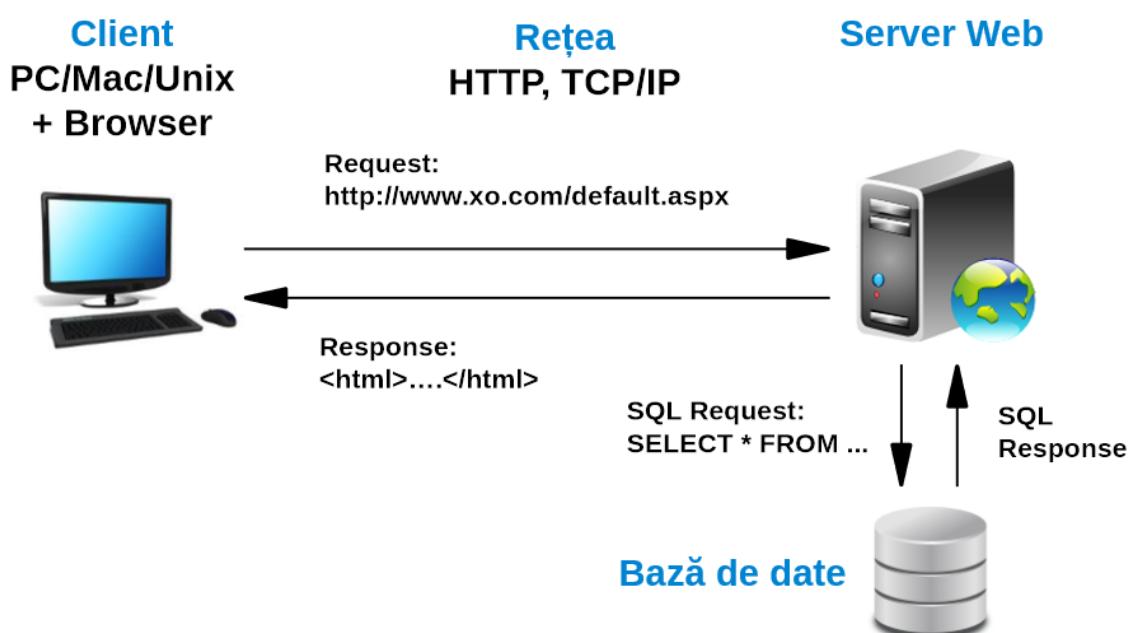
Ce este o aplicatie Web.....	2
Arhitectura Web.....	2
Avantajele aplicatiilor Web .....	3
Introducere in ASP.NET .....	3
Framework-ul .NET .....	4
ASP.NET Core.....	5
Introducere in C# .....	5
Limbaj compilat vs limbaj interpretat.....	6
Ciclul de viata al unei pagini Web.....	6

## Ce este o aplicatie Web

O aplicatie web este o aplicatie care ruleaza intr-o arhitectura Client-Server bazata pe: **protocolul HTTP, TCP/IP; browser Web; Server Web.**

Aplicatiile web sunt execute intr-un browser web si implementate folosind tehnologii precum: PHP, ASP, PYTHON, HTML, CSS, JAVASCRIPT, etc.

## Arhitectura Web



## Avantajele aplicatiilor Web

- Sunt independente de sistemul de operare
- Nu necesita instalare
- Actualizari foarte usor de facut deoarece modificarile se fac intr-un singur loc – pe server, ele propagandu-se pentru toti utilizatorii (in cazul aplicatiilor client-server clasice, interfata cu utilizatorul este asigurata prin intermediul unui program client instalat pe calculatorul fiecarui utilizator, orice modificare necesitand reinstalarea aplicatiei pentru fiecare utilizator in parte)

## Introducere in ASP.NET

- **ASP.NET** este un framework Web, open source, conceput si dezvoltat de Microsoft
- Este utilizat pentru a dezvolta aplicatii si servicii web
- Ofera o integrare foarte buna a codului HTML, CSS, JAVASCRIPT
- Este construit pe baza **CLR (Common Language Runtime)** – ruleaza **cod compilat** si permite utilizatorilor sa scrie cod folosind **orice limbaj .NET**

### Ce este CLR – Common Language Runtime?

Se ocupa de executia programelor C#. Atunci cand este compilat un program C# rezultatul compilarii **nu este un cod executabil**. In locul acestuia se produce un fisier care contine un tip de cod apropiat de codul masinii, numit limbaj intermediu sau pe scurt **IL – Intermediate Language**.

Prin intermediul unui compilator denumit **JIT – Just in Time**, CLR transforma codul intermediar în cod executabil.

## Framework-ul .NET

- Este compatibil cu peste 20 de limbaje diferite, cele mai populare fiind C#, C++, Visual Basic, F#
- Pună la dispozitie o colecție impresionantă de clase, organizate în biblioteci
- Este construit din două entități importante:

### 1. Common Language Runtime (CLR)

- mediul de execuție al programelor fiind cel care se ocupă cu managementul și execuția codului scris în limbi specifice .NET

### 2. Base Class Library

- Este biblioteca de clase .NET
- Acoperă o arie largă a necesităților de programare, incluzând **interfața cu utilizatorul, conectarea cu baza de date, accesarea datelor**

## ASP.NET Core

- Este noul framework creat de Microsoft
- A fost conceput pentru a functiona indiferent de sistemul de operare (Windows, MAC, Linux), fiind astfel mult mai flexibil si rapid
- ASP.NET Core este un framework nou, nefiind o continuare a framework-ului ASP.NET 4.6
- ASP.NET Core aduce in plus securitate, scade costurile si imbunatateste performanta
- Se pot dezvolta foarte multe tipuri de aplicatii: desktop, web, mobile, machine learning, micro-servicii, jocuri
- Hostare mult mai rapida in cloud

## Introducere in C#

- Este un limbaj compilat
- Este un limbaj orientat pe obiecte
- Permite dezvoltarea de aplicatii industriale, durabile
- A fost conceput ca un concurent pentru limbajul Java
- Este derivat al limbajului C++

## Limbaj compilat vs limbaj interpretat

**Limbaj compilat** – codul scris, numit cod sursa, este translatat de catre compilator intr-un cod apropiat de nivelul masinii, numit **cod executabil**. Atunci cand aplicatia trece de compilare fara erori de sintaxa se va produce codul executabil, iar aplicatia va putea fi rulata. (exemplu limbaje compilate: C, C++, Java, C#).

**Limbaj interpretat (la rulare)** – cu ajutorul unui interpretor specific limbajului, fiecare linie de cod este interpretata chiar in momentul rularii, fiind preschimbata imediat in cod masina si executata (exemplu limbaje interpretate: PHP, Ruby, Python)

## Ciclul de viata al unei pagini Web

Paginile ASP.NET ruleaza pe server-ul Web Microsoft IIS (Internet Information Server). In urma prelucrarii pe server rezulta o pagina web HTML, care este trimisa catre browser.

**Ciclul de viata al unei pagini Web ASP.NET** are urmatorii pasi:

- **Page request (accesarea paginii)** – acest pas se intampla inaintea ciclului de viata, atunci cand o pagina este ceruta serverului
- **Start** – in acest stadiu se incarca proprietatile paginii, cum ar fi request-ul si raspunsul si se identifica tipul acestora (**GET – cerere resurse, POST – trimiterea de informatii catre server**)
- **Initialization (initializare)** – in acest pas se initializeaza directivele si controalele si se aplica codul din Master Page

- **Load (incarcarea)** – in aceasta faza daca cererea este de tip **postback**, controalele sunt incarcate cu informatii
- **Evenimentele Postback** – daca cererea este de tip postback se executa codul aferent. Dupa executia codului se aplica sistemele de validare
- **Rendering (ex: afisarea paginii)** – in acest pas se construieste pagina finala pe server, care va fi afisata in browser
- **Unload (eliberarea memoriei)** – dupa ce pagina a fost trimisa utilizatorului, resursele alocate pentru aceasta sunt eliberate

# Dezvoltarea Aplicatiilor Web utilizand ASP.NET Core MVC

## Curs 2

---

### Cuprins

Introducere in C# .....	2
Structura unui program .....	2
Variabile si Tipuri de date.....	7
Conversii de tip .....	8
Nullable .....	11
Instructiuni de control .....	11
Array-uri.....	13
Conventii de nume - Naming conventions .....	14

# Introducere in C#

C# este unul dintre cele mai populare limbaje de programare, dezvoltat de Microsoft, care ruleaza pe .NET Framework.

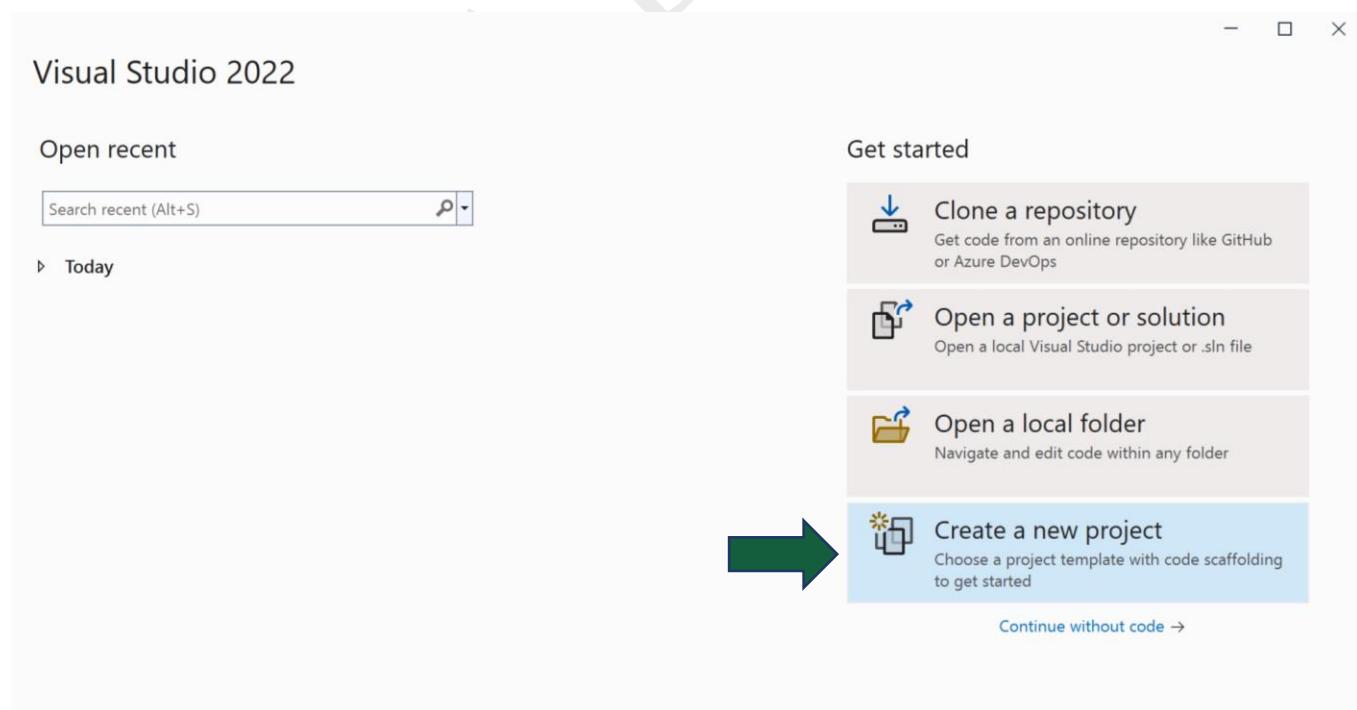
Este folosit pentru a dezvolta o gama variata de aplicatii, de la aplicatii web, mobile, desktop, pana la jocuri, servicii web, VR, etc.

## Structura unui program

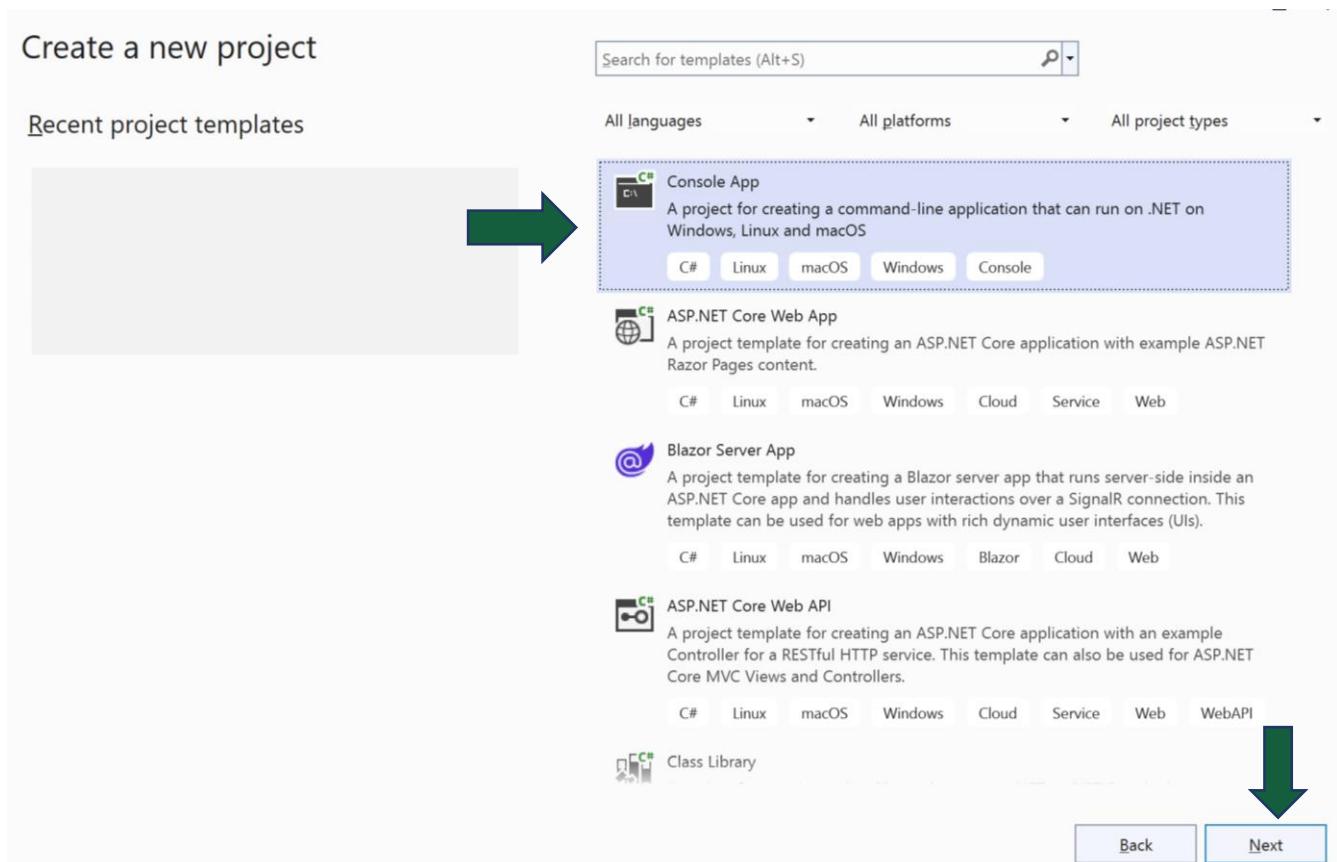
Pentru a dezvolta un program scris in C#, vom utiliza Visual Studio 2022 (**VEZI** Laborator 1 – instalare VS 2022).

Pentru inceput se creeaza un proiect folosind urmatoarele proprietati:

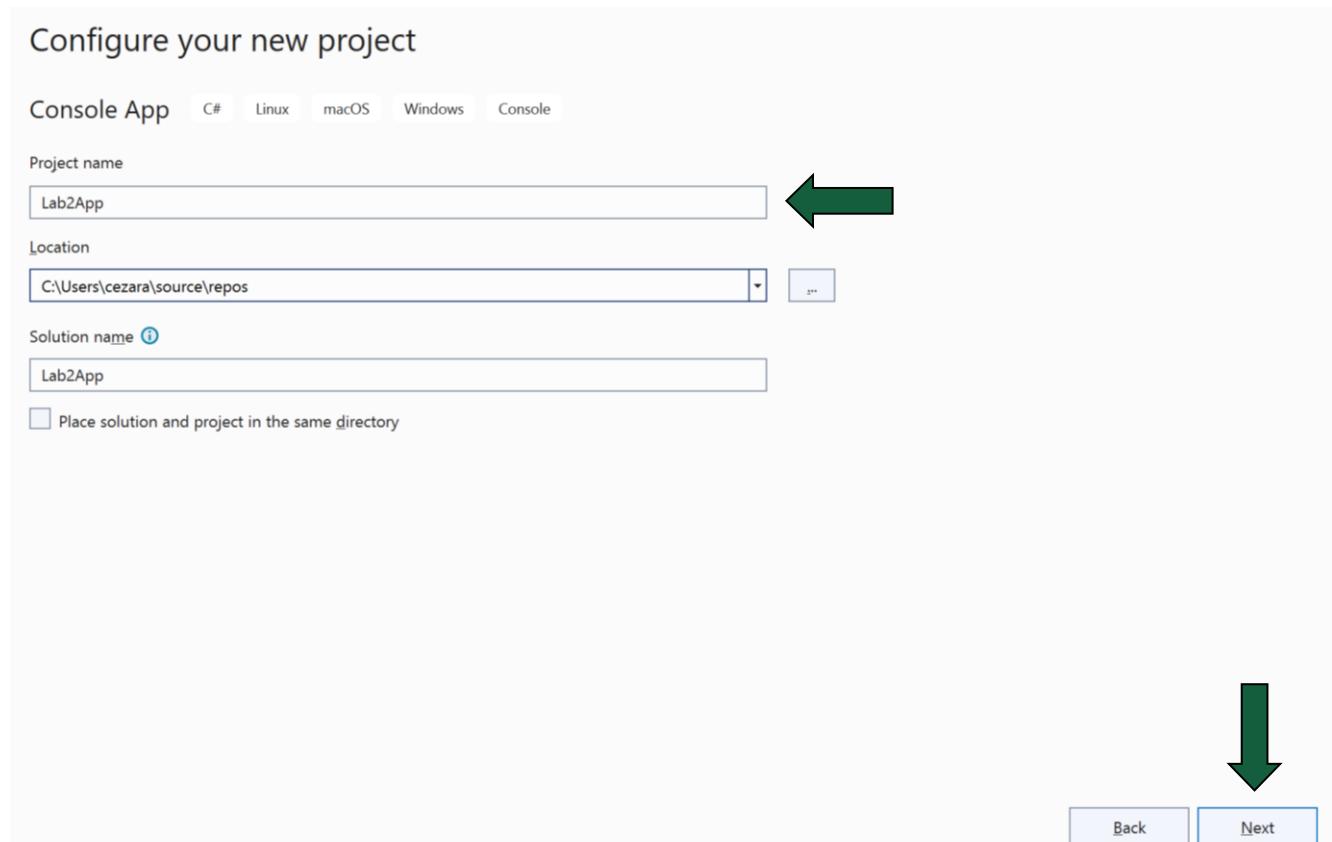
### PASUL 1:



## PASUL 2:

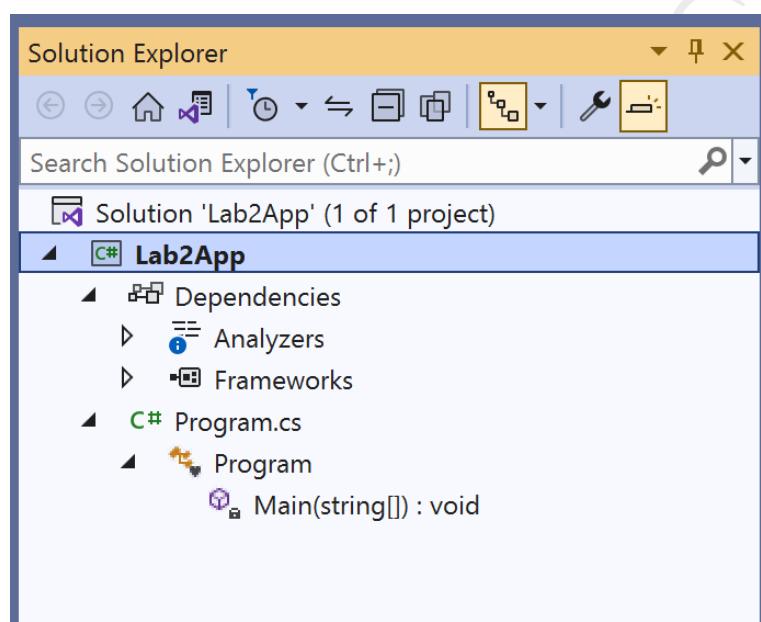
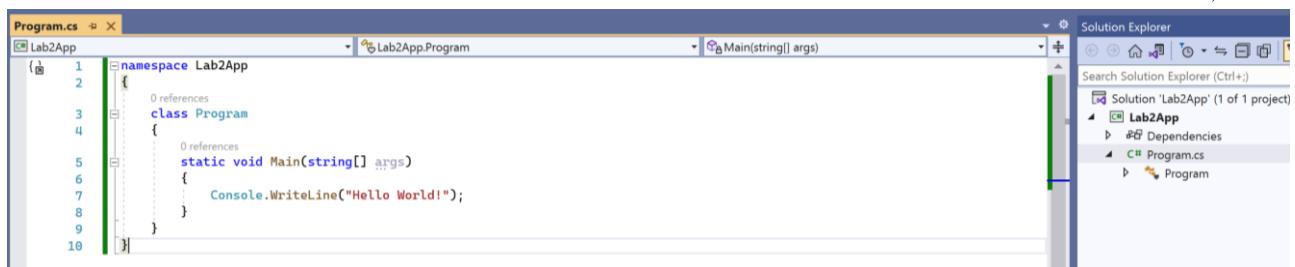


## PASUL 3:



## PASUL 4:

In fisierul **Program.cs** vom scrie cel mai simplu program in C#. Se va afisa in consola mesajul “Hello World!”

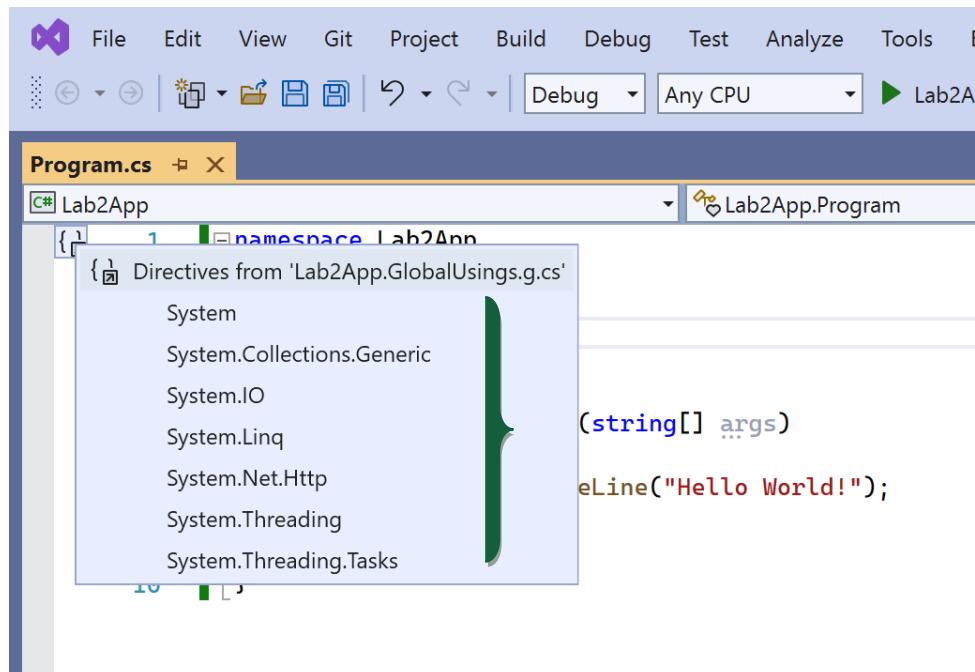


In momentul crearii proiectului, acesta a fost numit **Lab2App**.  
Denumirea proiectului devine automat **namespace**.

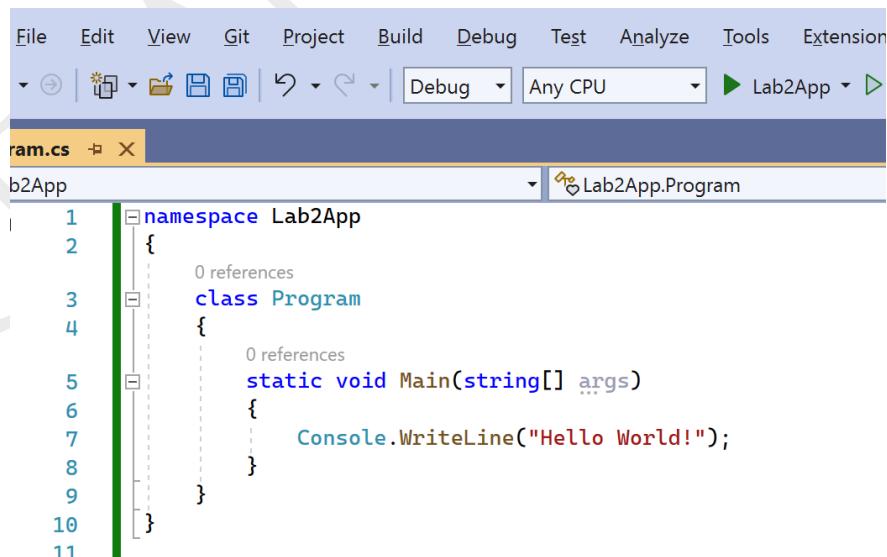
Un **namespace** – este o colectie de clase. Namespace-ul Lab2App contine clasa **Program**. Atunci cand scriem cod intr-un namespace, avem acces la toate clasele definite in el.

Daca se doreste utilizarea unei clase dintr-un alt namespace, atunci acel namespace trebuie importat.

Pentru import se utilizeaza **using**. Visual Studio are incluse cateva astfel de directive.



Directiva **System** – ofera acces la **toate clasele de baza**. In cazul de fata, System ofera acces la clasa Console care la randul ei contine metoda **WriteLine**.



Clasa **Program** – defineste metoda **Main**. Comparativ cu Java, unde clasa care contine metoda Main trebuie sa se numeasca tot Main, in cazul lui C# clasa poate avea orice denumire.

**static void Main (string[] args) – este metoda principală a programului**, metoda care se apeleaza prima.

### **⚠️ OBS:**

- C# este case sensitive (**VEZI** mai jos – [naming conventions](#))
- Orice declaratie sau expresie se incheie cu ;
- Executia programelor incepe intotdeauna cu metoda Main
- Comentariul pe o singura linie se include folosind //
- Comentariile pe mai multe linii se includ folosind /\*\*/

## **Variabile si Tipuri de date**

O **variabila** – este un nume pe care il ia o zona de memorie. Ulterior acel spatiu de memorie poate fi manipulat prin intermediul denumirii variabilei.

Fiecare variabila are un anumit **tip de date**, care determina dimensiunea zonei de memorie.

### **Tipuri de date in C#**

#### **Cele mai utilizate tipuri de date sunt:**

- int – numere intregi fara virgula
- double – numere intregi cu virgula
- char – stocheaza cate un caracter
- string – stocheaza text
- bool – stocheaza valori true sau false
- object – tipul obiect este clasa de baza pentru toate tipurile de date in C#. Tipurilor obiecte li se pot atribui valori de orice tip.

```
// TIPURILE DE DATE

int nr = 100;
string str = "Acesta este un text";
double d = 12.35;
char c = 'a';
bool b = true;
object obj = 100;

Console.WriteLine("Numarul este " + nr);
Console.WriteLine("Stringul este: " + str);
Console.WriteLine("Numarul in virugla mobila este: " + d);
Console.WriteLine("Caracterul este: " + c);
Console.WriteLine("Valoarea de adevar este: " + b);
Console.WriteLine("Obiectul este: " + obj);
```

## Conversii de tip



Conversiile de tip se impart in doua categorii:

- **implicită** – compilatorul C# convertește automat un tip de date în alt tip de date. În general tipuri de date care ocupă o zonă mai mică de memorie, cum este tipul int, sunt convertite automat în tipuri de date care ocupă o zonă mai mare de memorie.

```
// CONVERSII IMPLICITE

int nrInt = 10;

// Metoda GetType() preia tipul de date
Type tipNrInt = nrInt.GetType();

// Conversie implicită
double nrDouble = nrInt;

// Se preia tipul
Type tipNrDouble = nrDouble.GetType();

// Afisare valori inainte de conversie
Console.WriteLine("nrInt value: " + nrInt);
Console.WriteLine("nrInt Type: " + tipNrInt);

// Afisare valori după conversia implicită
Console.WriteLine("nrDouble value: " + nrDouble);
Console.WriteLine("nrDouble Type: " + tipNrDouble);
```

**Compilatorul a convertit implicit tipul int in double, fara pierdere de informatie.**

- **explicite** – in cazul in care se doreste conversia unui tip care ocupa o zona mai mare de memorie, intr-un tip care are o dimensiune mai mica a memoriei

```
// CONVERSII EXPLICITE

double nDouble = 25.123;

// Conversie explicita
int nInt = (int)nDouble;

// Afisarea valorii inainte de conversie
Console.WriteLine("Valoarea inainte de conversie a fost: " +
+ nDouble);

// Afisarea valorii dupa conversie
Console.WriteLine("Valoarea dupa conversie este: " +
nInt);
```

- Se poate utiliza si conversia folosind **Parse()**. Aceasta se utilizeaza cu precadere in cazurile in care se convertesc tipuri de date care nu sunt compatibile. De exemplu, int si string

**Sintaxa:** `int.Parse(parametru);`

```
// CONVERSIE UTILIZAND PARSE()

string st = "100";

// tipul de date
Type tip1 = st.GetType();

// Se converteste tipul string in int
int x = int.Parse(st);
Type tip2 = x.GetType();

Console.WriteLine("Valoarea initiala a fost: " + st);
Console.WriteLine("A avut tipul: " + tip1);

Console.WriteLine("Noua valoare dupa conversie este: " + x);
Console.WriteLine("Valoarea dupa conversie are tipul: " + tip2);
```

- **Conversii folosind clasa Convert** – clasa pune la dispozitie numeroase metode pentru a converti orice tip de date intr-un alt tip de date -> **ToBoolean()**, **ToChar()**, **ToDouble()**, **ToInt16()**, **ToString()**

```
// CONVERSII FOLOSIND CLASA CONVERT

int num = 25;
Console.WriteLine("Valoare de tip int: " + num);

// Se converteste valoarea int in stringul "25"
string strConvert = Convert.ToString(num);
Console.WriteLine("Valoarea dupa conversie " +
strConvert);
Console.WriteLine("Tipul dupa conversie: " +
strConvert.GetType());

// Conversie in Double
Double doubleConvert = Convert.ToDouble(num);
Console.WriteLine("Valoarea dupa conversie " +
doubleConvert);
Console.WriteLine("Tipul dupa conversie: " +
doubleConvert.GetType());
```

## Nullable

**Nullable** reprezinta in C# mai multe tipuri de date. Aceste tipuri de date pot lua valori dintr-un interval de valori sau pot fi null.

Sintaxa pentru declararea unui tip de date nullable este urmatoarea:

```
<tipul_de_date>? <nume_variabila> = null;
```

**Exemplu:**

```
// NULLABLE
int? num1 = null;
int? num2 = 45;

Console.WriteLine("Valorile sunt: {0}, {1}", num1, num2);
```

## Instructiuni de control

```
// INSTRUCTIUNI DE CONTROL
```

```
// if
if (Conditie)
{
    // se executa daca conditia este true
}

// if else
if (Conditie)
{
    // se executa daca conditia este true
}
else
{
    // se executa daca conditia este false
}
```

```

//if else --- else
if (Conditie1)
{
    // se executa daca conditie1 este true
}
else if (Conditie2)
{
    // se executa daca conditie2 este true
}
else if (Conditie3)
{
    // se executa daca conditie3 este true
}
else
{
    // se executa daca nicio conditie din cele anterioare
nu se indeplineste
}

// if imbricat (nested)
if (Conditie1)
{
    // se executa daca conditie1 este true
    if (Conditie2)
    {
        // se executa daca conditie2 este true
    }
}

// while loop
while (Conditie)
{
    statement(s);
}

// for loop
for (init; conditie; increment)
{
    statement(s);
}

```

```
// do...while loop
do
{
    statement(s);
} while(Conditie);
```

## Array-uri

**Un array** este utilizat pentru a stoca o colectie de date de acelasi tip.  
Fiecare element din array este accesat prin indexul sau.

Sintaxa de declarare a unui array:

**datatype[] numeArray;**

```
// ARRAY

int[] n = new int[10];

for (int i = 0; i < 10; i++)
{
    n[i] = i + 1;
}

for (int i = 0; i < 10; i++)
{
    Console.WriteLine("Element[{0}] = {1}", i, n[i]);
}
```

## Conventii de nume - Naming conventions

- Se utilizeaza **PascalCase** pentru:
  - Clase
  - Constructor
  - Metode
- Se utilizeaza **camelCase** pentru:
  - Variabile
  - Argumentele metodelor

# Dezvoltarea Aplicatiilor Web utilizand ASP.NET Core MVC

## Laborator 2

---

### EXERCITII:

1. Sa se creeze un nou proiect (**VEZI Curs 2**)
2. Sa se implementeze exemplele studiate in cursul 2.
3. Sa se creeze un nou proiect de tipul Console App, numit Laborator2 si sa se implementeze urmatoarea problema.

Se citeste un numar intreg n de la tastatura. Sa se verifice daca numarul este palindrom si sa se afiseze un mesaj corespunzator (ex: “Numarul n este / nu este palindrom”). Se vor trata si cazurile particulare. De exemplu, toate numerele formate dintr-o singura cifra sunt palindrom.

Problema se va rezolva in urmatoarele variante:

1. direct in Main avand urmatoarea structura:

```
class Program
{
    static void Main(string[] args)
    {
        ...
    }
}
```

2. se va crea o metoda Palindrom astfel:

```
class Program
{
    void Palindrom (int n)
    {
        ...
    }

    static void Main(string[] args)
    {
        ...
    }
}
```

Numarul n se citeste de la tastatura utilizand **Console.ReadLine()**;

4. Se citeste un vector V cu n elemente numere naturale ( $n \leq 100$ ).

Verificati daca oricare doua elemente alaturate alterneaza ca paritate (adica au paritate diferita). Sa se afiseze un mesaj corespunzator.

**EX: n=5 si V=[12, 53, 8, 73, 2] -> Da**

# Dezvoltarea Aplicatiilor Web utilizand ASP.NET Core MVC

## Curs 3

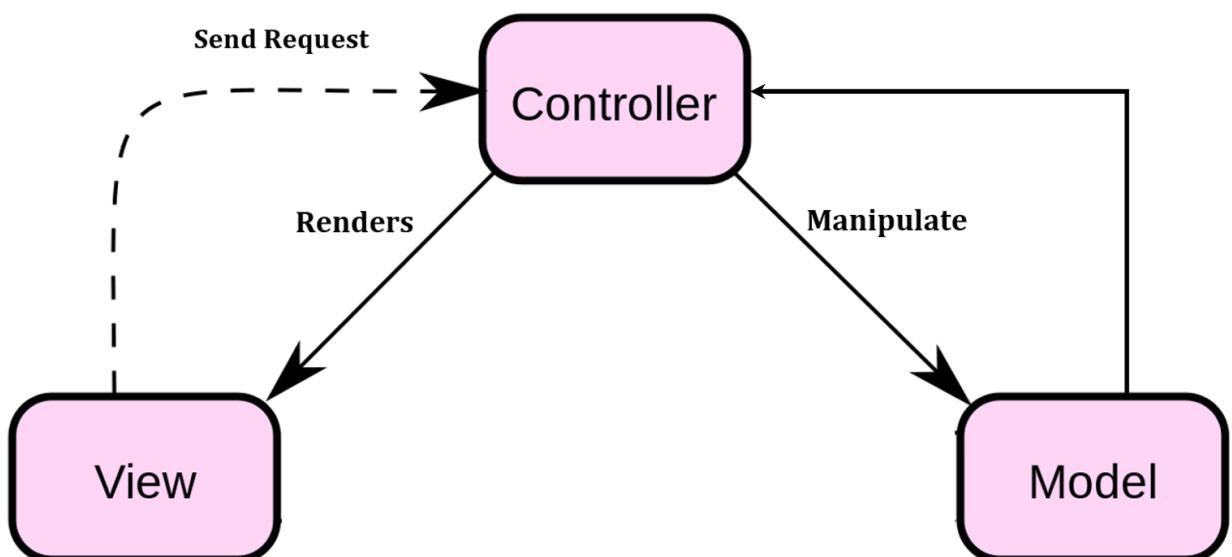
---

### Cuprins:

Arhitectura MVC .....	2
Model (Stratul business – prelucrarea datelor) .....	2
Controller .....	3
View (interfata cu utilizatorul).....	3
Crearea unei aplicatii in ASP.NET Core 6.0 (Visual Studio 2022) .....	4
Structura unui proiect MVC – Sistemul de fisiere.....	7
Sistemul de rutare.....	10
Exemple de implementare a rutelor: .....	18
Configurarea rutelor:.....	18
Definirea rutelor custom .....	24
Constrangerile parametrilor .....	26

## Arhitectura MVC

**Model-View-Controller (MVC)** este un model arhitectural utilizat în dezvoltarea aplicațiilor. Succesul modelului se datorează izolării logicii aplicatiei (business logic) față de interfața cu utilizatorul, rezultând o aplicație unde aspectul vizual și nivelele inferioare ale regulilor de business sunt mai ușor de modificat, fără a afecta alte nivale.



### Model (Stratul business – prelucrarea datelor)

**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.

## Controller

**Controller-ul** este reprezentat de clase, fiind componenta care controleaza accesul la aplicatie.

In Controller:

- sunt procesate requesturile HTTP
- se citesc datele introduse de utilizator
- are loc procesarea prin trimitera datelor catre Model - unde se executa operatiile
- se trimit raspunsul catre View

Asadar, Controller-ul comunica cu Modelul si cu View-ul.

Fiecare Controller contine asa numitele **Action-uri (Actions)** reprezentate de metode. Aceste metode sunt publice, se definesc in interiorul unui Controller si sunt accesate in momentul in care are loc un request prin intermediul unei rute.

## View (interfata cu utilizatorul)

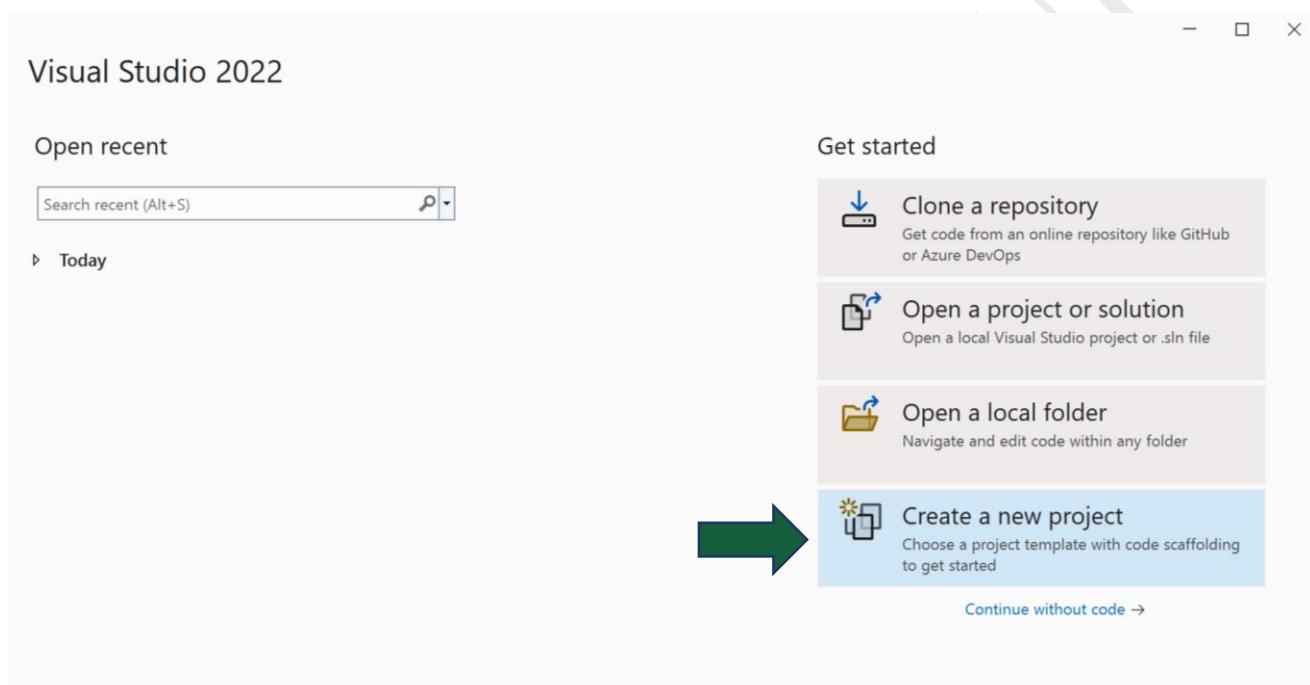
**View-ul** reprezinta interfata cu utilizatorul, fiind componenta arhitecturii MVC cu care utilizatorii interactioneaza prin intermediul browser-ului.

In View se afiseaza datele, adica inregistrarile din baza de date si informatiile generate de aplicatie.

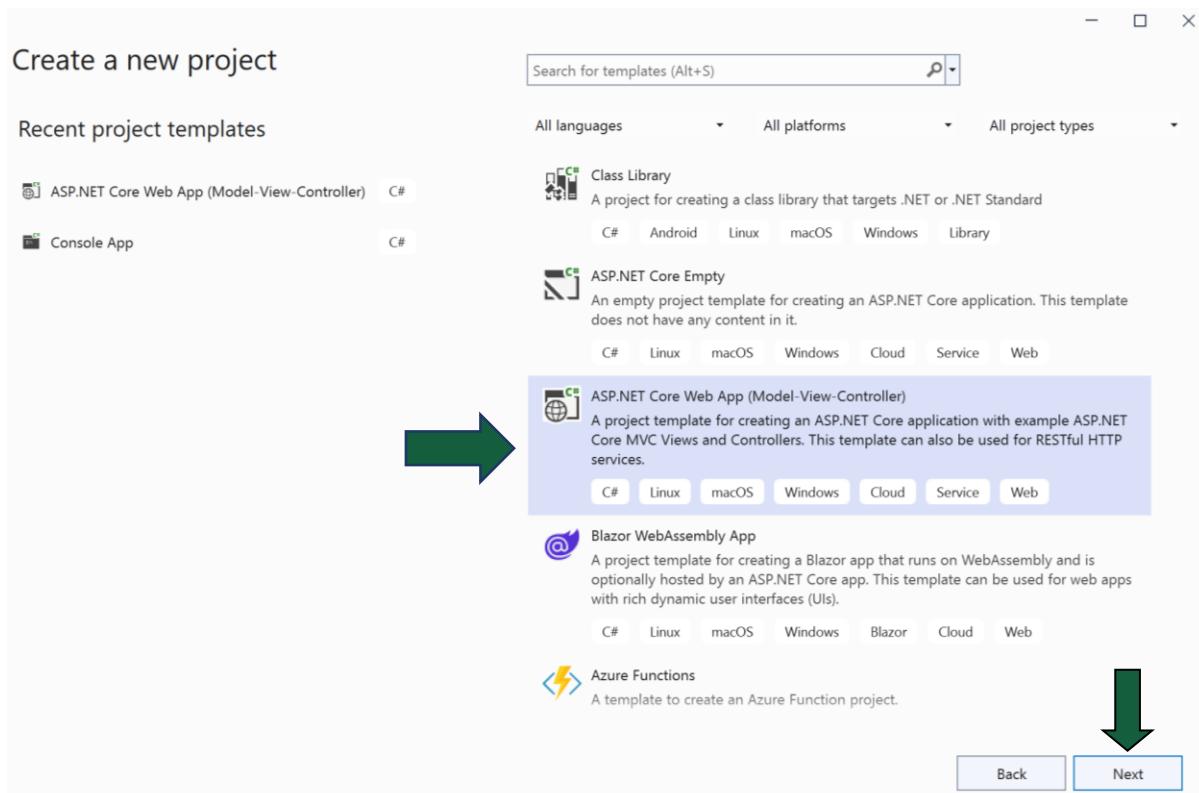
View-ul poate contine HTML static sau chiar HTML trimis din Controller (HTML dinamic). In cadrul arhitecturii MVC, View-ul comunica doar cu Controller-ul, iar cu Modelul indirect tot prin intermediul Controller-ului.

# Crearea unei aplicatii in ASP.NET Core 6.0 (Visual Studio 2022)

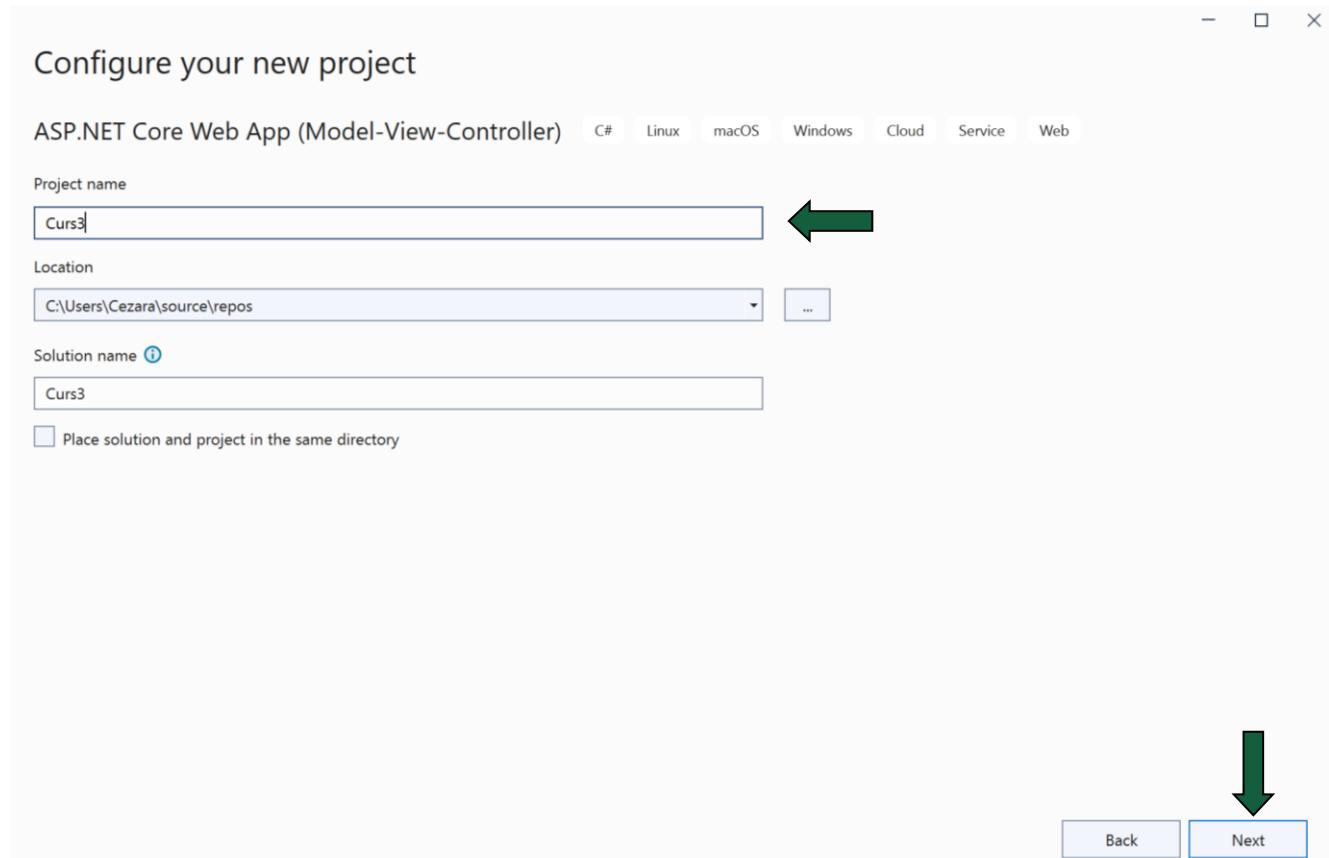
## PASUL 1:



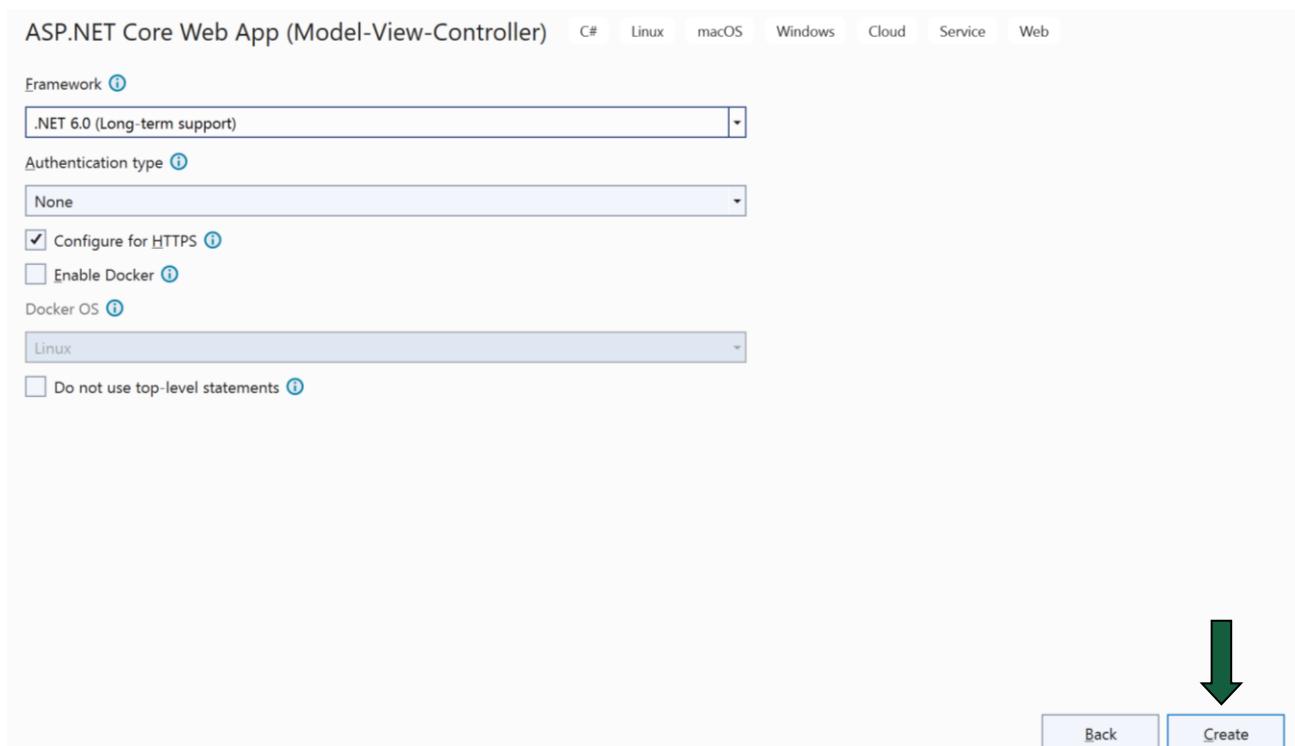
## PASUL 2:



## PASUL 3:

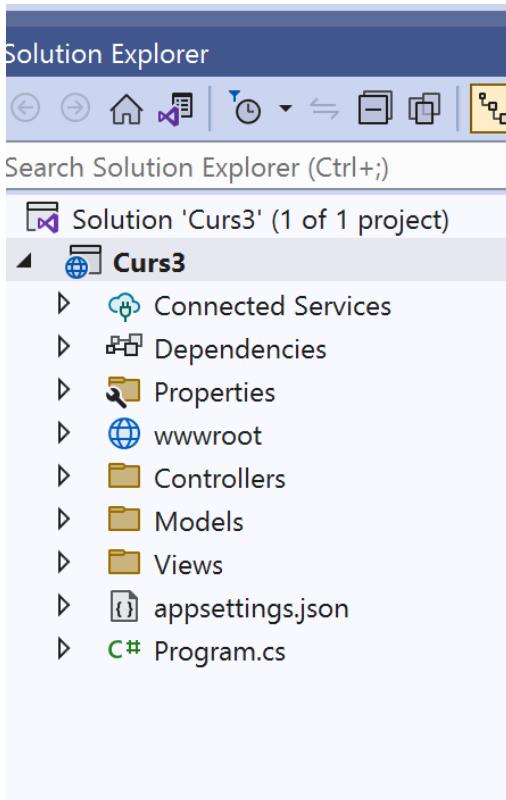


## PASUL 4:



## Structura unui proiect MVC – Sistemul de fisiere

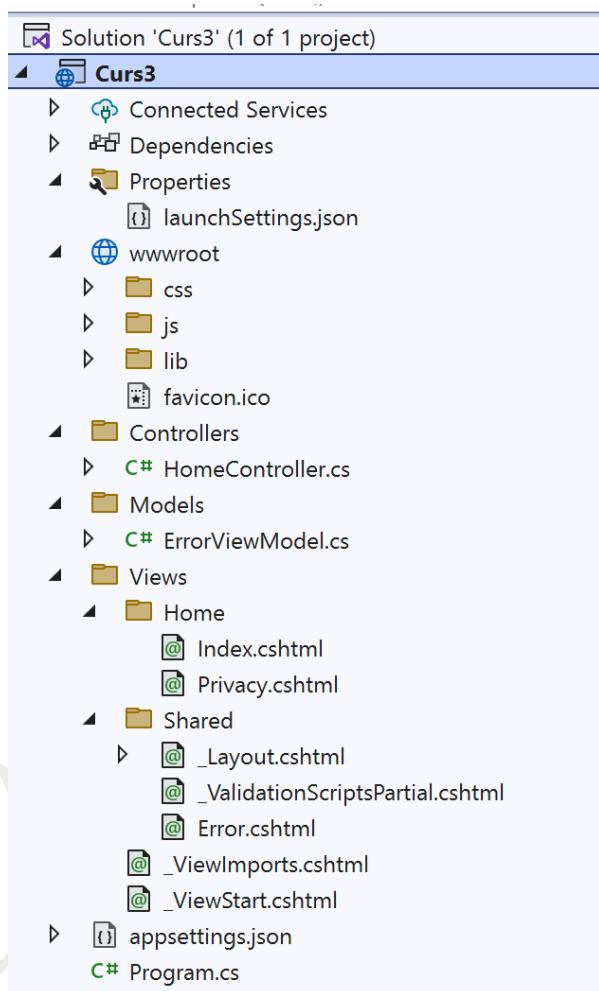
In imaginea urmatoare este prezentata structura unui proiect MVC realizat in ASP.NET Core 6.0. Structura proiectului este reprezentata de sistemul de fisiere si foldere pe care le contine aplicatia.



- **wwwroot** – contine fisierele statice precum librarii (ex: Bootstrap), imagini, scripturi (css, js)
- **Controllers** – include toate fisierele de tip Controller. Acestea sunt fisiere care contin cod c# si au extensia .cs. MVC impune ca numele tuturor controller-elor sa contin la final cuvantul Controller
- **Models** – folderul contine modelele aplicatiei
- **Views** – folderul contine fisierele de tip View (interfata cu utilizatorul) ale aplicatiei
- **appsettings.json** – in acest fisier se afla setarile pentru configuratia aplicatiei. Se utilizeaza pentru stocarea detaliilor de conectare la baza de date
- **Program.cs** – este punctul de pornire al aplicatiei care se acceseara imediat dupa ce aplicatia este rulata. De asemenea, in acest fisier se configureaza modulele aplicatiei: domeniul

aplicatiei (host), serverul web (IIS, Nginx, etc), modulul de autentificare, etc.

In imaginea urmatoare se pot observa toate folderele si fisierele existente intr-un proiect nou creat.



Tot sistemul de fisiere o sa fie studiat pe rand in cursurile urmatoare.

## Sistemul de rutare

ASP.NET a introdus termenul de **Routing** si implicit rutarea pentru a elimina necesitatea maparii fiecarui URL cu un fisier fizic, asa cum era necesar in versiunea anterioara de ASP.NET Web Forms, unde fiecare URL trebuia sa coincida cu un fisier .aspx.

**Rutele** reprezinta diferite URL-uri din aplicatie care sunt procesate de un anumit Controller si de o anumita metoda pentru generarea unui raspuns catre client. Framework-ul ASP.NET MVC invoca diverse clase de tip Controller, impreuna cu diferite metode ale acestora, in functie de URL-ul cerut de client.

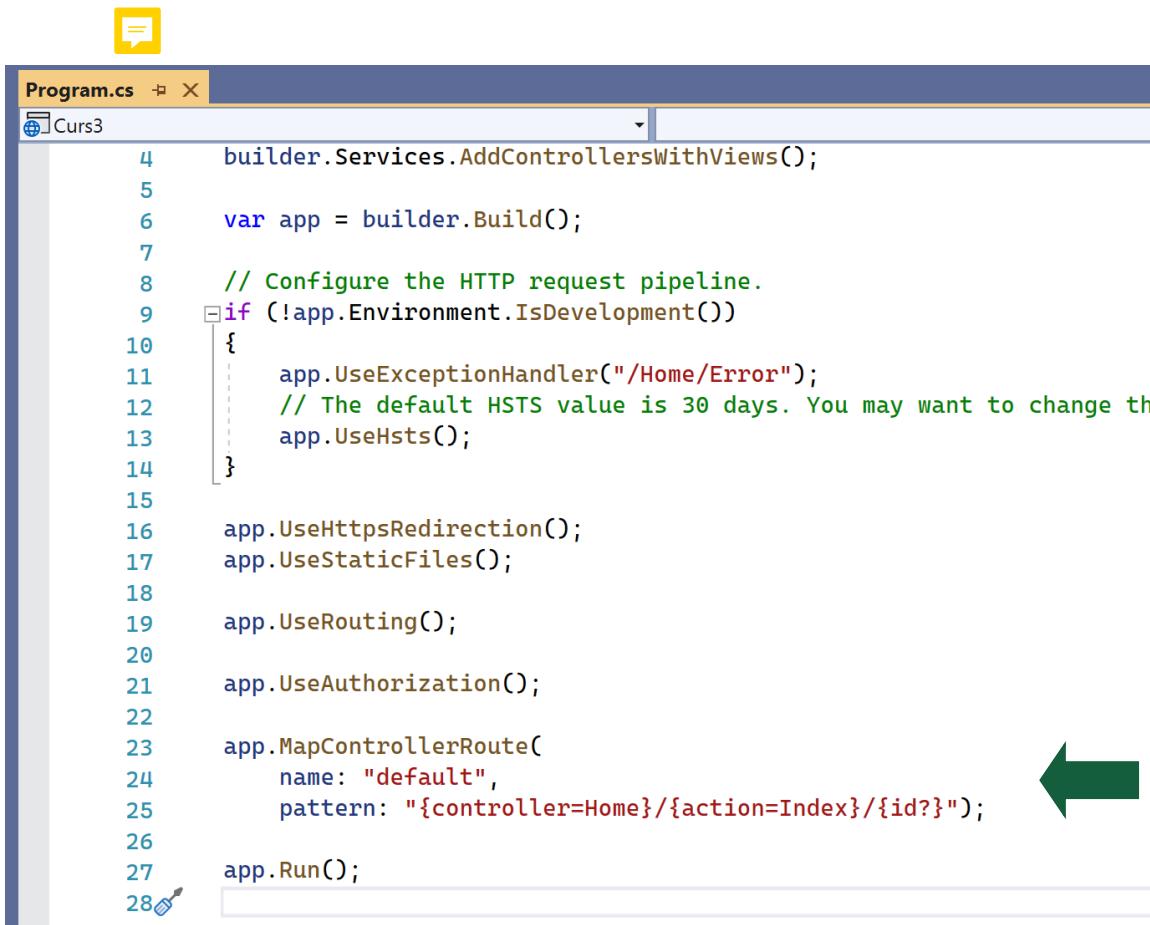
Astfel, pentru a accesa o anumita pagina, este necesar ca pentru aceasta sa existe o ruta definita, cat si un Controller care are o metoda(Action) care sa raspunda acestei resurse.

Formatul de baza al rutelor in ASP.NET este urmatorul:

**/{{NumeController}}/{{NumeActiune}}/{{Parametrii}}**

Rutele se definesc in clasa **Program.cs**

In **Program.cs** se poate observa ruta default:



```

1  builder.Services.AddControllersWithViews();
2
3  var app = builder.Build();
4
5  // Configure the HTTP request pipeline.
6  if (!app.Environment.IsDevelopment())
7  {
8      app.UseExceptionHandler("/Home/Error");
9      // The default HSTS value is 30 days. You may want to change this.
10     app.UseHsts();
11 }
12
13 app.UseHttpsRedirection();
14 app.UseStaticFiles();
15
16 app.UseRouting();
17
18 app.UseAuthorization();
19
20 app.MapControllerRoute(
21     name: "default",
22     pattern: "{controller=Home}/{action=Index}/{id?}");
23
24 app.Run();
25
26
27
28

```

Metoda **UseRouting()** se ocupa de configurarea rutelor in aplicatiei.

Definirea unei rute are urmatorul format:

```

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

```

Se observa ca variabila **app**, care este de tipul clasei **WebApplication** (clasa care se ocupa de configurarea HTTP pipeline si a rutelor), ofera mai multe metode necesare definirii rutelor.

Metoda **MapControllerRoute** adauga ruta default si primeste ca argumente 2 parametri:

- **name**: care reprezinta numele rutei (ex: `name: "default"`)
- **pattern**: care reprezinta modelul URL-ului sau segmentele acestuia (ex: `pattern: {controller=Home}/{action=Index}/{id?}`).

In plus sunt definite detaliiile dupa cum urmeaza:

- **controller** – primeste ca valoare numele controller-ului care sa raspunda la aceasta ruta
- **action** - primeste ca valoare numele metodei din controller care sa raspunda la aceasta ruta
- pentru fiecare parametru adaugat in ruta, defineste **tipul parametrilor**, daca acestia sunt **necesari** sau **optionali** sau se pot seta valorile implice

In exemplul anterior se defineste ruta `/Home/Index/{id}` care este procesata de Controller-ul **HomeController** prin metoda **Index**. Parametrul **id** este optional si poate fi omis din URL. Fiind urmat de “**?**” -> `{id?}`, inseamna ca este un parametru optional.

In acest sistem de rutare, pot fi mai multi parametri delimitati prin caracterul “/“.

Asadar, in acest caz, metoda **app.MapControllerRoute()** mapeaza endpoint-uri cu pattern-ul:

```
{controller=Home}/{action=Index}/{id?}
```

**Endpoint-ul** din ASP.NET Core este reprezentat de Controller, fiind acea unitate responsabila cu procesarea request-urilor.

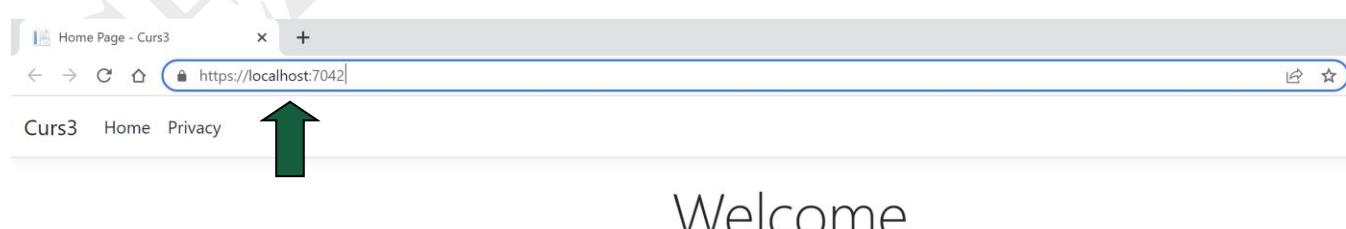
Privind exemplul anterior putem spune ca de fiecare data cand aplicatia primeste un URL care sa se potriveasca cu pattern-ul, atunci o sa se acceseze Controller-ul numit Home si Action-ul (metoda) Index. Acest lucru se intampla doar in cazul in care nu este specificat un alt Controller sau o alta metoda. In cazul in care sunt specificate explicit alte endpoints, atunci aceleia o sa fie accesate.

In cazul in care aplicatia este accesata fara segmentele necesare in cadrul URL-ului, adica se cere pagina principala a aplicatiei “/”, framework-ul ASP.NET MVC va raspunde in mod implicit cu metoda “**Index**” din Controllerul “**Home**”.

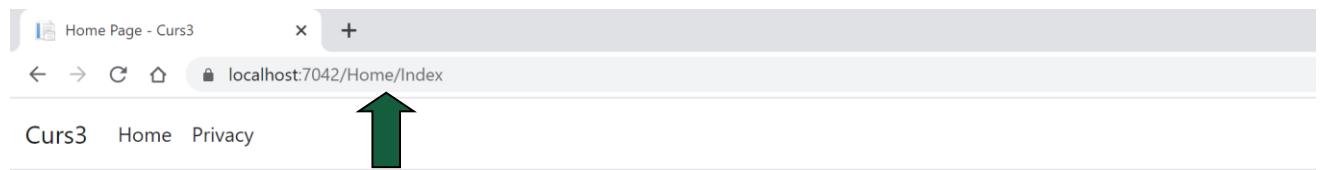
In concluzie, pattern-ul **{controller=Home}/{action=Index}/{id?}** se potriveste cu toate URL-urile de forma: **/**, **/Home**, **/Home/Index**, **/Students/Index**, **/Students/Index/5**, **/Students/Afisare**, etc.

In exemplul urmator se poate observa existenta Controller-ului **HomeController**. In cadrul clasei HomeController exista mai multe metode (Actions). Se poate observa prezenta metodei **Index()**. In momentul in care exista o metoda, trebuie sa existe si un View asociat. View-ul trebuie sa se numeasca identic cu metoda, deci **Index.cshtml**. Privind configuratia default a rutei, putem observa ca aceasta este ruta default, adica ruta care se acceseaza prima si afiseaza catre utilizatorul final interfata care se afla in Index.cshtml.

Dupa rulare se acceseaza in browser ruta default:

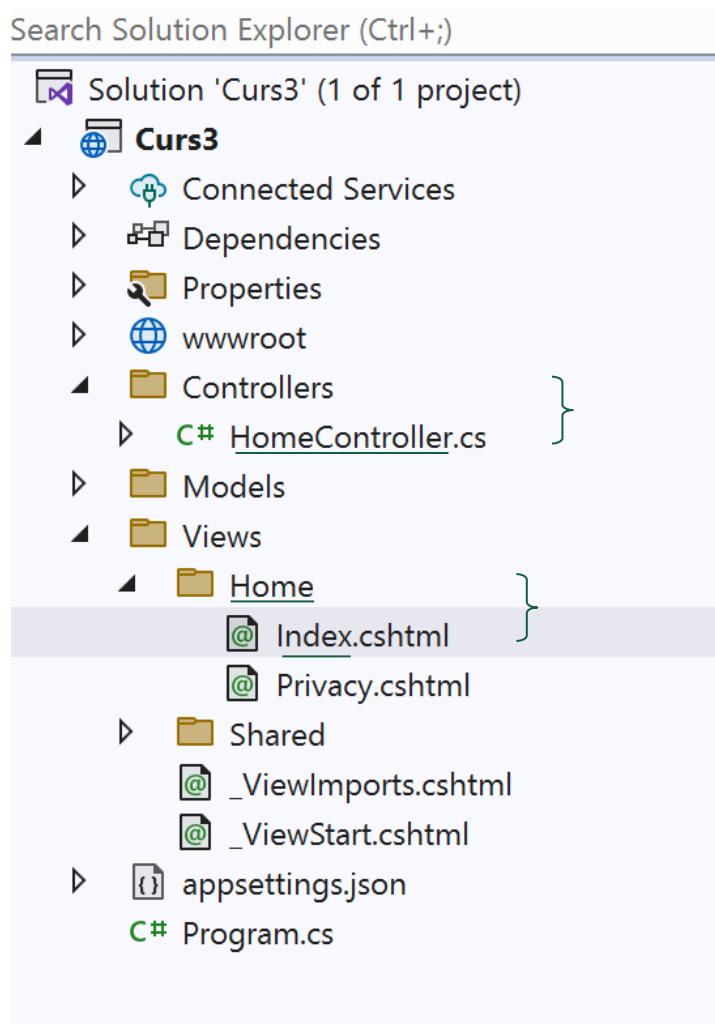


In cazul in care se acceseaza pagina prin scrierea intregului URL, atunci ruta o sa aiba urmatorul format:



Welcome

In **Solution Explorer** se observa:



In **HomeController** există metoda Index:

```

Index.cshtml      HomeController.cs      Program.cs
Curs3            Curs3.Controllers.HomeController
1   using Curs3.Models;
2   using Microsoft.AspNetCore.Mvc;
3   using System.Diagnostics;
4
5   namespace Curs3.Controllers
6   {
7       public class HomeController : Controller
8       {
9           private readonly ILogger<HomeController> _logger;
10
11          public HomeController(ILogger<HomeController> logger)
12          {
13              _logger = logger;
14          }
15
16          public IActionResult Index()
17          {
18              return View();
19          }
20

```

The diagram illustrates the flow from the HomeController.cs file to the Index.cshtml view. A large green arrow points down from the controller code to the view. Three smaller green arrows point from specific lines in the controller code to the corresponding lines in the view: one from the 'Index()' method signature to the 'Index()' method in the view, another from the 'return View();' statement to the 'View()' call in the view, and a third from the 'logger' parameter declaration to the '\_logger' field in the view.

In **View** -> Folderul Home (asociat controller-ului HomeController) -> Index.cshtml (pagina pentru codul html asociata metodei Index din Controller)

```

Index.cshtml      HomeController.cs      Program.cs
1 @{
2     ViewData["Title"] = "Home Page";
3 }
4
5 <div class="text-center">
6     <h1 class="display-4">Welcome</h1>
7     <p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web
8     </div>
9

```

The diagram illustrates the flow from the Index.cshtml view back to the HomeController.cs file. A large green arrow points up from the view to the controller. Two smaller green arrows point from the 'ViewData["Title"]' assignment in the view to the 'ViewData' property in the controller's constructor: one from the assignment itself and another from the 'Title' key.

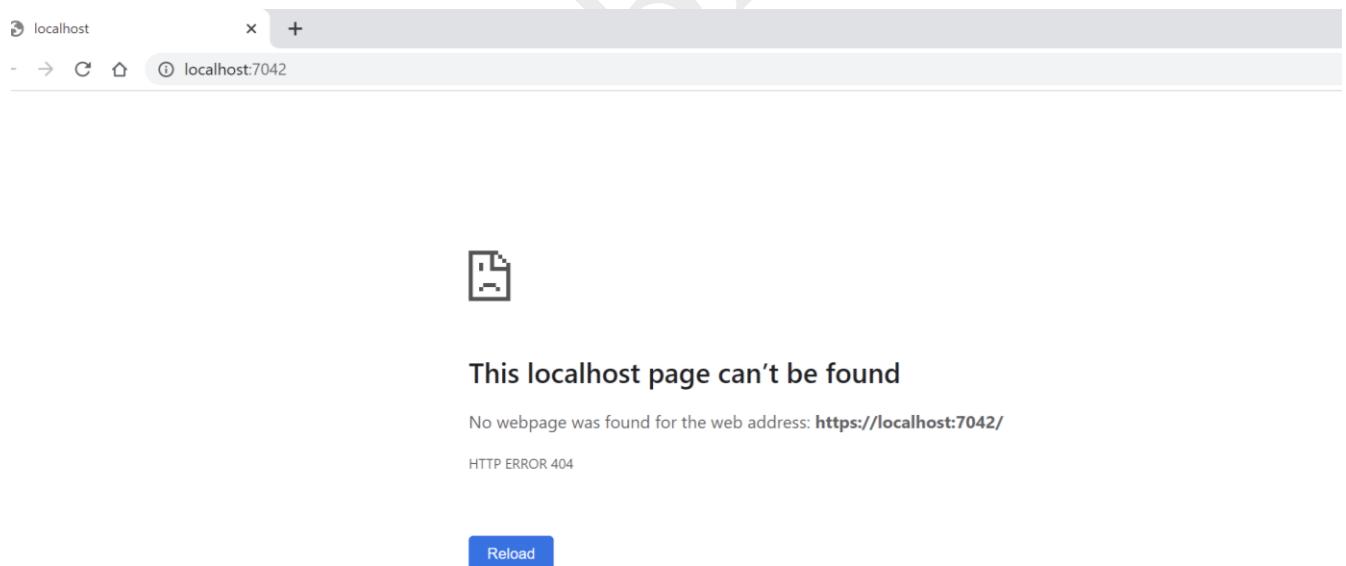
In cazul in care definitia rutei default nu contine si valorile implice, atunci nu se pot accesa paginile aplicatiei.

De ex: se elimina ruta deja definita si se adauga urmatoarea configuratie:

```
/*
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.MapControllerRoute(
    name: "default",
    pattern: "{controller}/{action}");
```

In momentul rularii apare mesajul **HTTP ERROR 404** deoarece nu gaseste pagina. Acest lucru se intampla din cauza faptului ca sistemul de rutare nu poate asocia ruta cu niciun Controller.

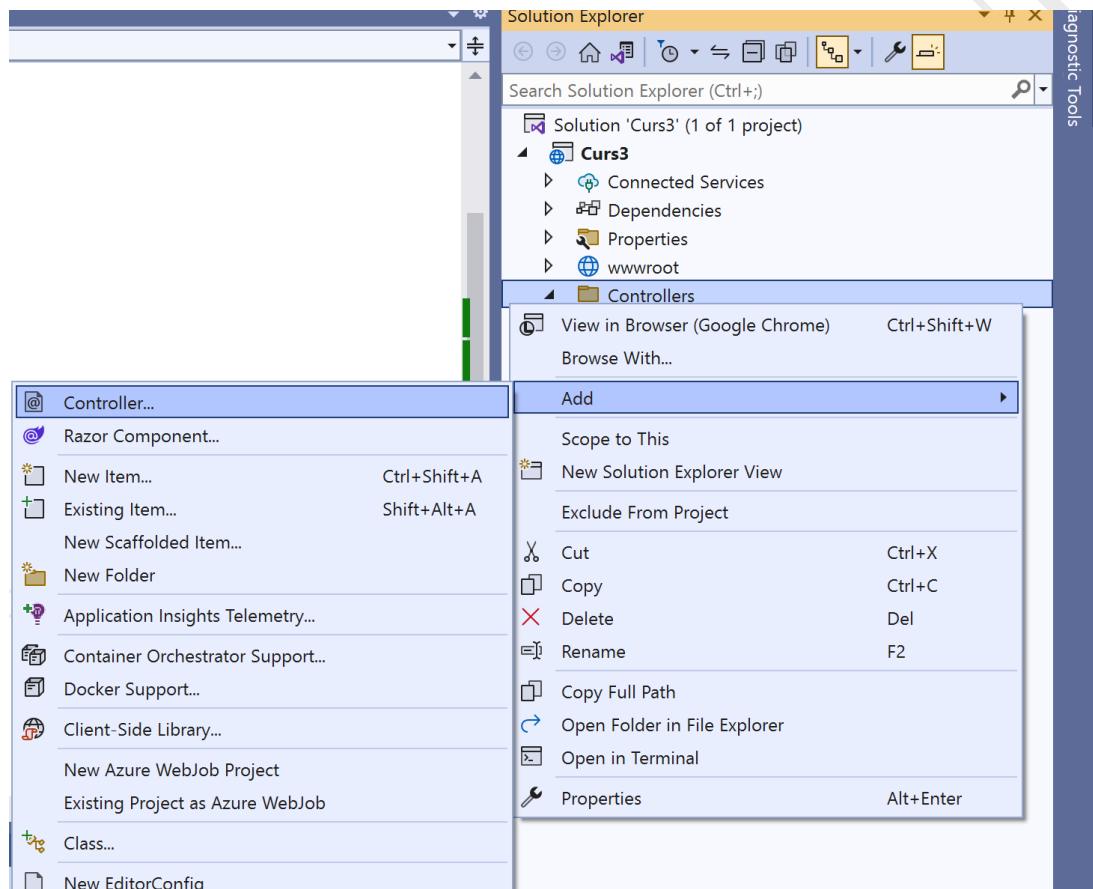


## Exemple de implementare a rutelor:

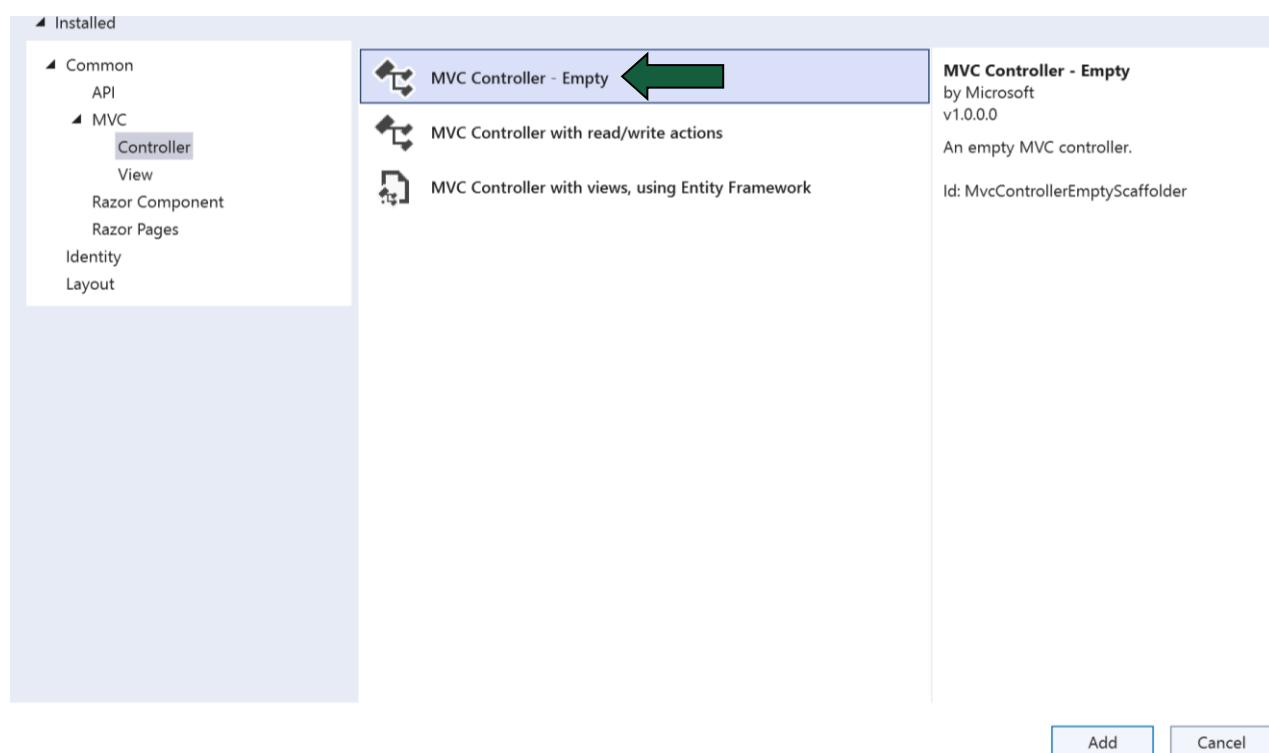


### Configurarea rutelor:

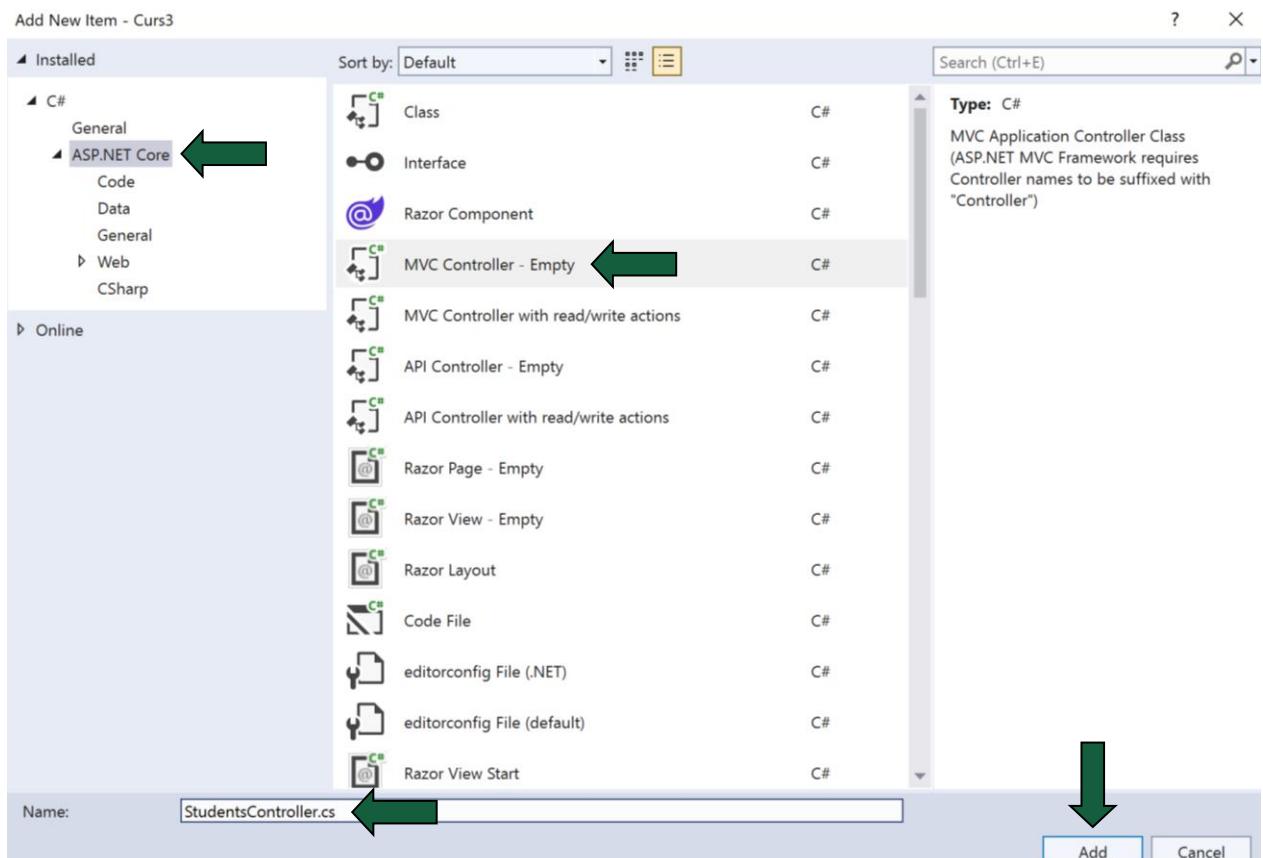
Pentru exemplele urmatoare se va crea un nou Controller, numit **StudentsController**. Click dreapta pe folderul Controller -> Add -> Controller.



Se selecteaza MVC Controller – Empty:



Se modifica numele noului Controller:



Pentru afisarea unui text, o sa se utilizeze metoda Index din cadrul Controller-ului **StudentsController**. In acest moment nu se va utiliza niciun View asociat.

```
public class StudentsController : Controller
{
    public string Index()
    {
        string response = "Hello World";
        return response;
        //return View();
    }
}
```

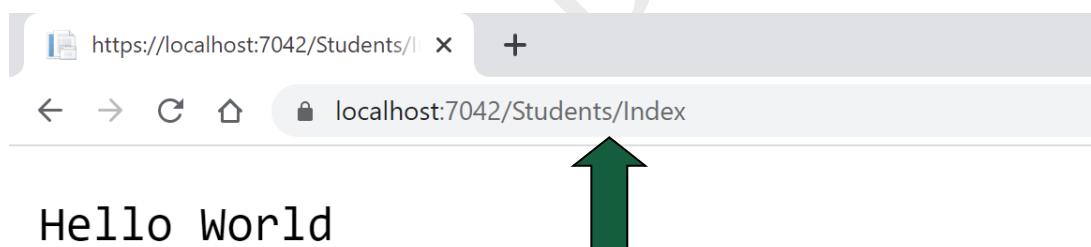
Tipul returnat de metoda Index a fost schimbat in **string** pentru a putea afisa in browser un simplu text. Astfel, valoarea de return devine valoarea variabilei **response**.

Dupa rulare, mesajul o sa fie afisat in browser, accesand ruta **/Students/Index** (**NumeController/NumeActiune**). In momentul accesarii URL-ului, request-ul se trimit aplicatiei, dupa care se incearca maparea URL-ului cu o configuratie de ruta prezenta in Program.cs.

Astfel, pattern-ul o sa primeasca noile valori:

- controller = Students
- action = Index
- id = este optional si poate sa lipseasca

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```



In continuare se defineste o ruta dupa cum urmeaza:

- ruta o sa contine doi parametri – name si id
- parametrul name o sa aiba o valoare implicita “World!”;
- parametrul id o sa fie optional

Metoda Index din controller o sa afiseze mesajul “Hello World!”, folosind valoarea parametrului **name** provenita din cadrul rutei.

Definirea rutei:



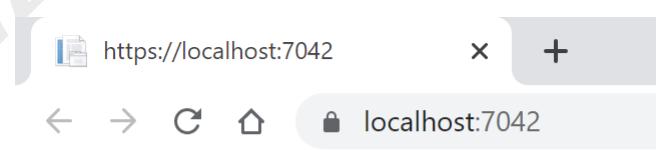
```
app.MapControllerRoute(
    name: "HelloWorld",
    pattern: "{controller=Students}/{action=Index}/{name=World!}/{id?}");
```

Implementarea metodei in Controller-ul StudentsController, metoda Index.

```
public string Index(string name, int? id)
{
    string response = "Hello " + name + " ";
    if (id != null)
    {
        response = response + "id = " + id;
    }

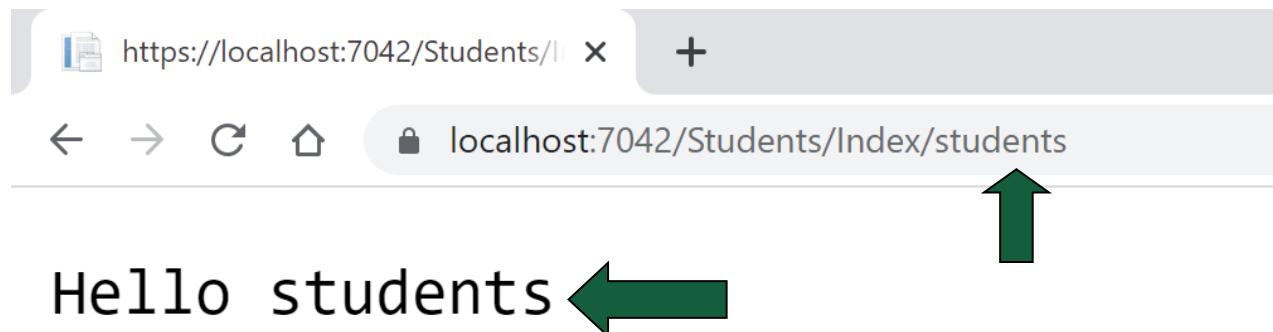
    return response;
    //return View();
}
```

In momentul in care se ruleaza aplicatia, fara a introduce segmentele URL-ului, se observa ca pentru variabila **name** s-a transmis valoarea acesteia implicita: **World!**, afisandu-se mesajul Hello World!

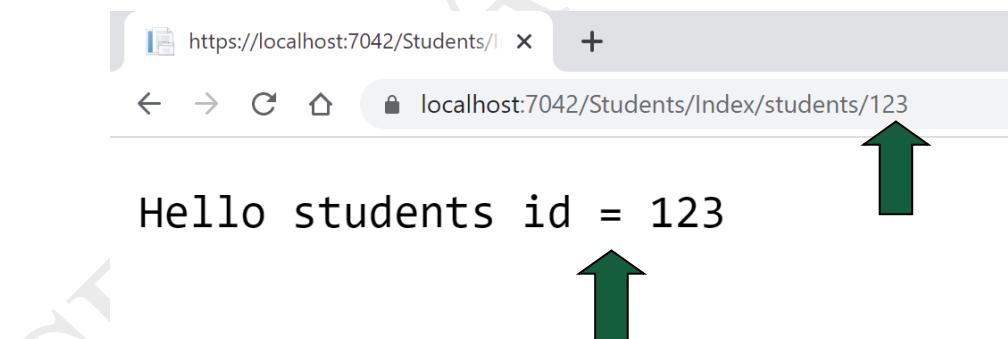


Hello World!

La adaugarea valorii pentru variabila **name** in URL, se observa cum aceasta a fost transmisa catre Controller si a fost afisata in pagina.



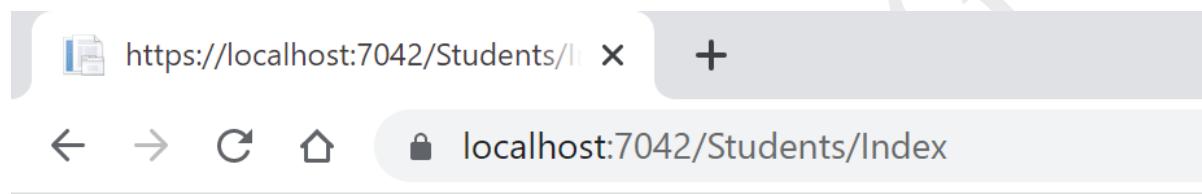
Cand parametrul optional **id** are valoare, se executa de cod specifica acestuia este executata si valoarea sa apare in raspunsul primit de la Controller:



## OBSERVATIE:

**⚠️** Ruta trebuie definita inaintea rutei default, deja existenta in fisierul Program.cs, deoarece rutele sunt interpretate in mod cascada (de sus in jos). Framework-ul utilizeaza prima configuratie din fisier care contine acelasi numar de parametrii ca ruta accesata din browser.

De exemplu, daca ruta default este definita inaintea rutei creata in exemplul anterior, iar URL-ul de accesare este **/Students/Index**, atunci configuratia rutei default o sa se potriveasca si vom avea un rezultat ca cel de mai jos. Nu se afiseaza si valoarea implicita a parametrului **name**.



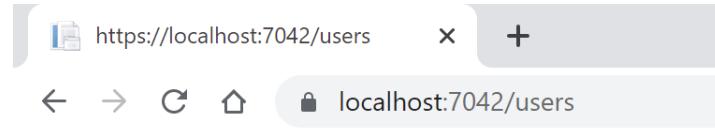
## Hello

## Definirea rutelor custom

Pentru fiecare **Controller** si **Actiune** in parte se pot defini si rute custom. De exemplu daca se doreste accesarea controller-ului Students si a metodei Index printr-un URL de forma: **/users** se poate implementa urmatoarea ruta:

```
app.MapControllerRoute(
    name: "Users",
    pattern: "users/{controller=Students}/{action=Index}/{name=world!!!}");
```

In varianta anterioara se poate accesa /Students/Index/name doar prin intermediul URL-ului: **/users** deoarece restul parametrilor vor prelua valorile implicite. Daca trebuie introdusa o alta valoare pentru parametrul **name**, atunci URL-ul trebuie sa fie: **users/Students/Index/abc**.



## Hello world!!!

Astfel, ruta **/users** a accesat controller-ul StudentsController, metoda Index cu paramentrul implicit name=world!!!

In acelasi mod se poate proceda si pentru mai multe elemente in ruta:

```
app.MapControllerRoute(
    name: "HomePage",
    pattern: "Home/Page/{controller=Home}/{action=Index}");
```



Definitia anterioara a rutei functioneaza si pentru URL-uri de tipul:

```
/Home/Page/Students/Index
/Home/Page/Users/Read
```

Adica functioneaza pentru orice alt nume de Controller si Actiune

Daca se doreste limitarea unei rute la o singura actiune, dintr-un singur controller, atunci se foloseste urmatoarea varianta:

```
app.MapControllerRoute(
    name: "HomePage",
    pattern: "Home/Page",
    defaults: new { controller = "Home", action = "Index" });
```

Acest lucru se intampla deoarece pattern-ul nu are niciun parametru configurabil, iar ruta o sa mapeze doar controller-ul Home si metoda Index.

De asemenea, varianta anterioara se foloseste si pentru cazul in care se doreste accesarea rutei doar printr-un URL custom. De ex: /**users**

```
app.MapControllerRoute(
    name: "Users2",
    pattern: "users/{name?}/{id?}",
    defaults: new { controller = "Students", action = "Index" });
```

### OBSERVATIE:

**⚠️** In momentul scrierii rutelor, dezvoltatorul trebuie sa se asigure ca nu exista ambiguitate intre definitiile acestora.

## Constrangerile parametrilor

Pentru a asigura un anumit tip de date sau un anumit format pentru parametrii transmisi catre Controller este necesara declararea unor constrangeri.

Exista mai multe tipuri de constrangeri: de tip, length, max, min, range, regex.

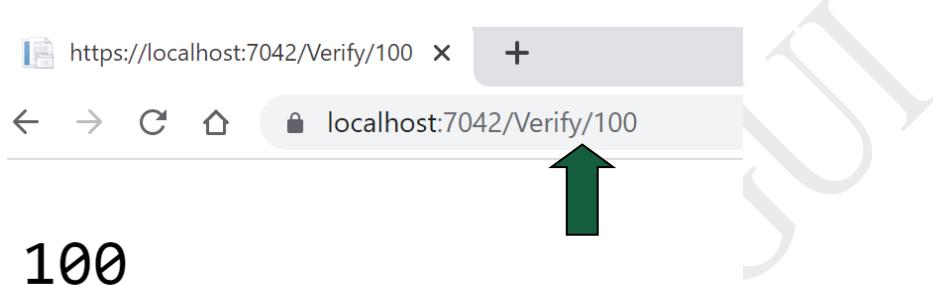
Un exemplu de constrangere este:

```
app.MapControllerRoute(
    name: "HomePage",
    pattern: "Verify/{id:range(10,100)}",
    defaults: new { controller = "Home", action = "Verify" });
```

Pentru un URL de tipul **/Verify/50** se cauta pattern-ul potrivit, dupa care se acceseaza parametrii din **defaults** -> controller-ul **Home** si metoda **Verify**. Daca parametrul **id** se afla in intervalul inchis 10, 100, atunci o sa acceseze ruta, iar in caz contrar o sa se afiseze 404 Not Found.

Pentru verificare se poate implementa in controller-ul Home, metoda Verify care afiseaza id-ul in pagina.

```
public int Verify(int id)
{
    return id;
}
```

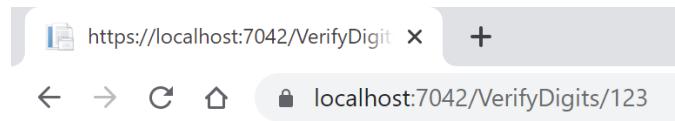


In cazul **expresiilor regulate** se configureaza ruta astfel:

```
app.MapControllerRoute(
    name: "HomePage",
    pattern: "VerifyDigits/{id:regex(\\d+)}",
    defaults: new { controller = "Home", action = "VerifyDigits" });
```

Ruta se acceseaza folosind URL-ul: /VerifyDigits/id, id-ul are o constrangere folosind o expresie regulata – se verifica daca este numar accesand apoi metoda VerifyDigits din controller-ul **Home**.

```
public string VerifyDigits(int id)
{
    return "VerifyDigits " + id;
}
```



VerifyDigits 123

# Dezvoltarea Aplicatiilor Web utilizand ASP.NET Core MVC

## Laborator 3

---

### EXERCITII:

#### Exercitiul 1:

Sa se creeze un nou proiect (**VEZI Curs 3 - Crearea unei aplicatii in ASP.NET Core 6.0**). In cadrul proiectului sa se adauge un nou Controller numit **ExamplesController** si sa se implementeze urmatoarele cerinte, creand metodele aferente in acest Controller (Actions):

1. Sa se adauge o ruta numita **Concatenare** care sa raspunda prin **/concatenare** urmata de 2 parametri de tip string. Sa se afiseze valorile lor concatenate.
2. Sa se adauge o ruta numita **Produs** care primeste 2 parametri de tip intreg. Al doilea parametru este optional. Sa se afiseze produsul celor 2 parametri. Daca al doilea parametru lipseste se va afisa mesajul “Introduceti ambele valori”. Ruta se va accesa prin ruta: **/produs/param1/param2**
3. Sa se adauge o ruta numita **Operatie** care primeste 3 parametri astfel: 2 de tip intreg si unul de tip string. Parametrul de tip string reprezinta o operatie (plus, minus, ori, div) efectuata intre parametrii de tip intreg. Sa se afiseze pe ecran toti parametrii impreuna cu operatia, cat si rezultatul acestora. In cazul in care un parametru lipseste se va afisa pe ecran mesajul “Introduceti parametrul 1/2/3” – in functie de parametrul care lipseste. De asemenea, toti parametrii sunt optionali. Ruta se va accesa prin URL-ul: **/operatie/param1/param2/op**  
Param op poate fi: “plus”, “minus”, “ori”, “div”

## Exercitiul 2:

Presupunem ca avem o aplicatie in care exista resursa Student. Sa se adauge un Controller – **StudentsController** si toate rutele asociate operatiilor de tip C.R.U.D. – CREATE, READ, UPDATE, DELETE.

De exemplu: in **StudentsController** o sa avem urmatoarele metode:

- **Index** – pentru afisarea tuturor studentilor
- **Show** – afisarea unui singur student, in functie de id-ul acestuia
- **Create** – crearea unui nou student
- **Edit** – editarea unui singur student
- **Delete** – stergerea unui singur student

Rutele trebuie sa contin toti parametrii necesari operatiilor.

De asemenea, mare atentie sa nu existe conflict intre rute.

## Explicatii:

**Index** – afisarea tuturor studentilor. Se vor realiza doua rute, in doua moduri.

**Varianta 1:** accesarea paginii in care se vor afisa toti studentii va avea loc folosind URL-ul: **/Students/Index**. Ruta o sa se numeasca **StudentsIndex**

**Varianta 2:** accesarea paginii va avea loc folosind URL-ul: **students/all**. Ruta o sa se numeasca **StudentsAll**

**Show** – afisarea unui singur student, in functie de id-ul acestuia. Se va realiza o ruta numita **StudentShow**, avand URL-ul: **students/{id}**. Id-ul este un parametru optional. In Controller se va afisa doar un mesaj: “Afisare student cu ID: ”

**Create** – crearea unui nou student. Se va realiza o ruta numita **StudentCreate**, avand URL-ul: **students/new**. In Controller se va afisa mesaj: “Creare student in baza de date”.

**Edit** – editarea unui singur student. Se va realiza o ruta numita **StudentEdit**, avand URL-ul: **students/{id}**. Id-ul este un parametru optional. In Controller se va afisa doar un mesaj: “Editare student cu ID: ”.

### Ce observati in acest caz?

**Delete** – stergerea unui singur student. Se va realiza o ruta numita **StudentDelete**, avand URL-ul: **students/{id}**. Id-ul este un parametru optional. In Controller se va afisa doar un mesaj: “Stergere student cu ID: ”

### Exercitiul 3:

Sa se creeze Controllerul **SearchController**. In cadrul acestuia sa se adauge urmatoarele metode: **NumarTelefon, CNP**.

Fiecare metoda reprezinta o pagina de cautare care primeste un parametru dupa care se face cautarea. Adaugati metodele impreuna cu toti parametrii lor, cat si rutele asociate impreuna cu urmatoarele constrangeri si afisati valorile parametrilor in pagina:

- Ruta pentru cautarea numarului de telefon este de forma: **search/telefon/{telefon}**. Ruta primește un singur parametru reprezentat de un sir de caractere de lungime = 10 caractere. Toate caracterele sunt cifre, sirul începe cu caracterul 0, după caracterul 0 trebuie să fie obligatoriu încă 9 cifre (<https://regexr.com/> ).

Ce observați în cazul în care numărul de telefon lipsește din request sau în cazul în care acesta are lungimi diferite? Aceste verificări se vor realiza la nivel de controller.

#### **De exemplu:**

```
if (telef.Length < 10)
    return "Numarul de telefon nu are suficiente cifre";
```

- Ruta CNP (/search/cnp/{cnp}) va primi un singur parametru reprezentat de un sir de caractere de lungime = 13. Toate caracterele sunt cifre. Sirul începe cu 1,2,5 sau 6; (<https://regexr.com/> ).

#### **Ce observați?**

# Dezvoltarea Aplicatiilor Web utilizand ASP.NET Core MVC

## Curs 4

---

### Cuprins

Controller .....	2
Ce este Controller-ul .....	2
Crearea unui proiect .....	3
Adaugarea unui nou Controller.....	4
Controller-ul implicit – HomeController .....	7
Actions .....	9
Structura unei metode .....	9
Raspunsul actiunilor – ActionResult .....	10
Parametrii unei actiuni .....	13
Selectori .....	13
ActionName .....	14
NonAction .....	15
ActionVerbs .....	15
Exemplu definire rute .....	17
Redirect in cadrul metodelor.....	19
Redirect .....	19
RedirectToRoute .....	19
RedirectToAction.....	20
RedirectPermanent/ RedirectToRoutePermanent/	
RedirectToActionPermanent.....	20
Returnare HTTP Status Code.....	21

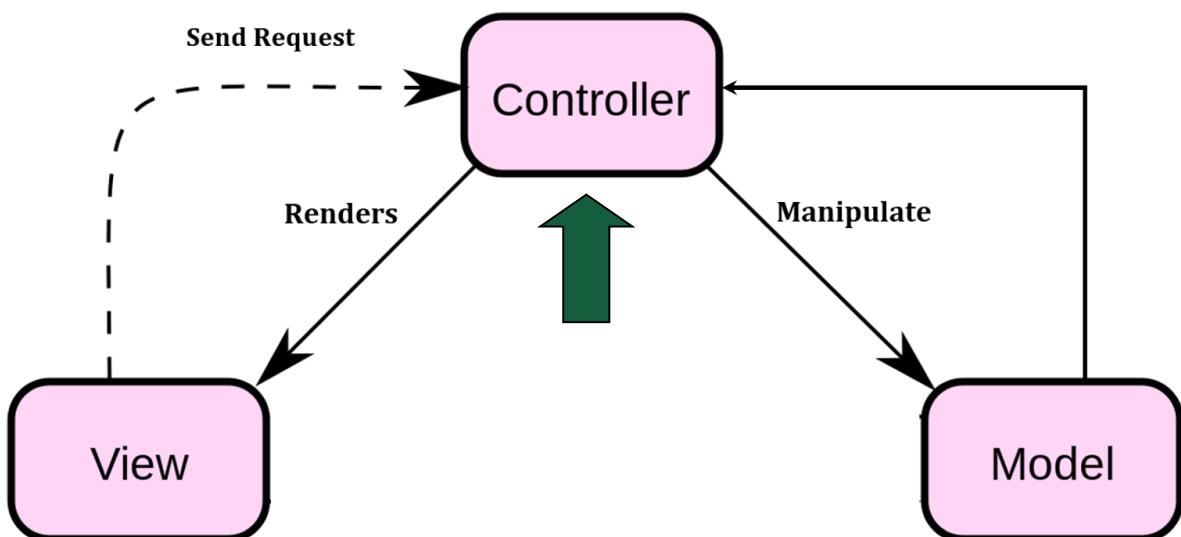
# Controller

## Ce este Controller-ul

In arhitectura MVC **Controller-ul** este componenta care proceseaza toate URL-urile aplicatiei. Controller-ul este o clasa, derivata din clasa de baza Microsoft.AspNetCore.Mvc. Aceasta clasa contine metode publice numite **Actiuni**. Metodele din Controller sunt responsabile pentru a procesa request-urile venite de la browser, pentru apelarea modelelor si procesarea datelor, cat si pentru a trimite raspunsul final catre utilizator prin intermediul browser-ului.

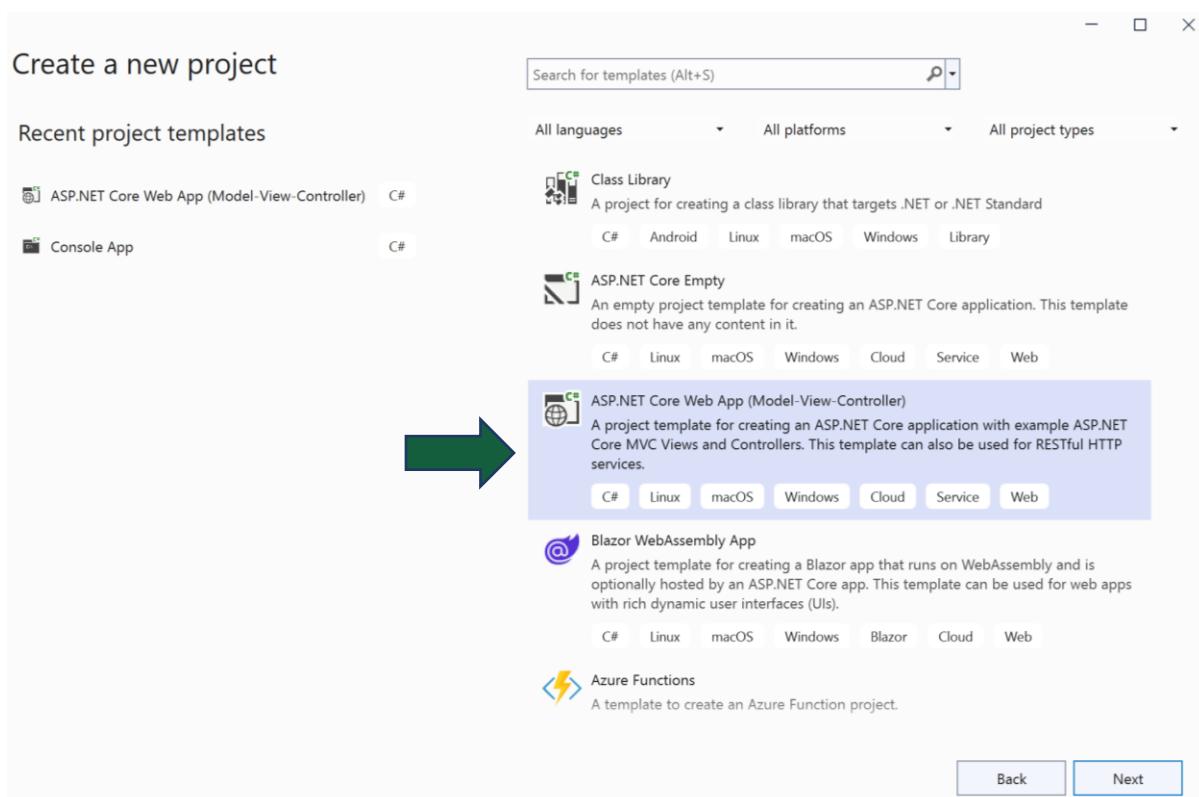
In ASP.NET MVC fiecare Controller este reprezentat de o clasa. Numele Controller-ului trebuie sa se termine in cuvantul **Controller**. De exemplu, Controller-ul pentru pagina Home se poate numi **HomeController**.

Controller-ele trebuie sa fie adaugate in cadrul folderului **Controllers** din proiectul ASP.NET.



## Crearea unui proiect

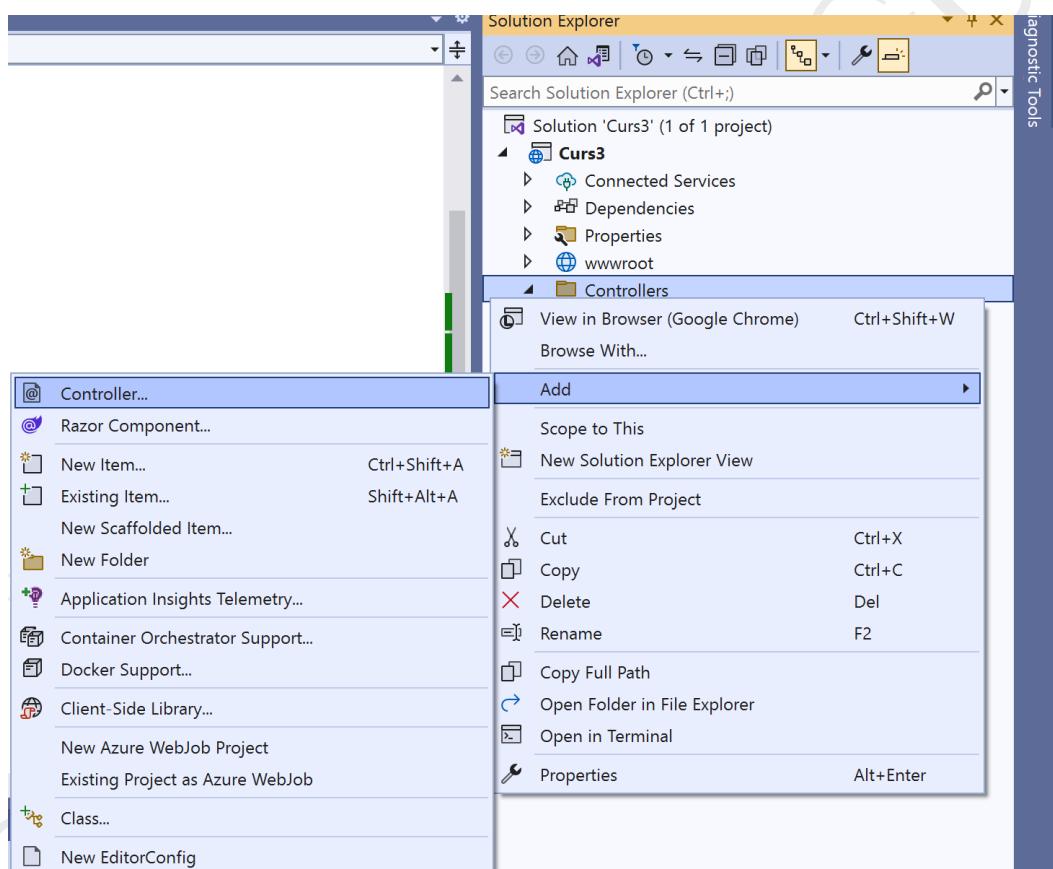
Pentru crearea unui nou proiect se utilizeaza optiunea ASP.NET Core Wep App (Model-View-Controller).



## Adaugarea unui nou Controller

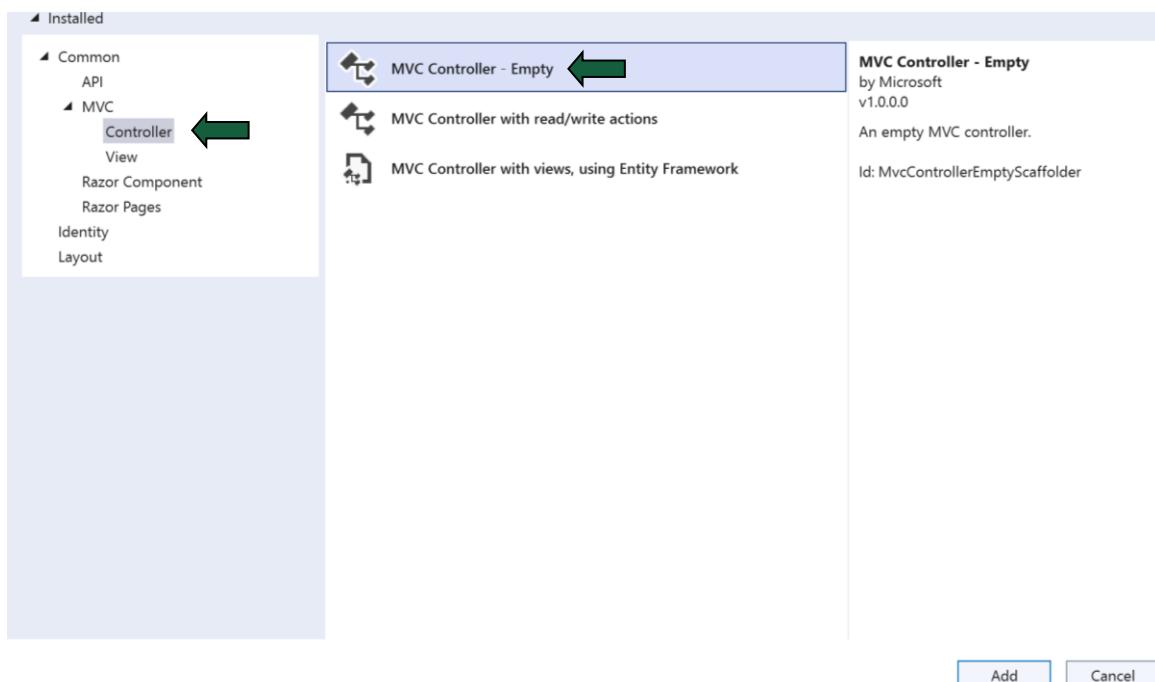
### Pas 1:

Pentru a adauga un Controller se procedeaza astfel: click dreapta pe folderul **Controllers** si din meniul **Add** se selecteaza **Controller**.



## Pas 2:

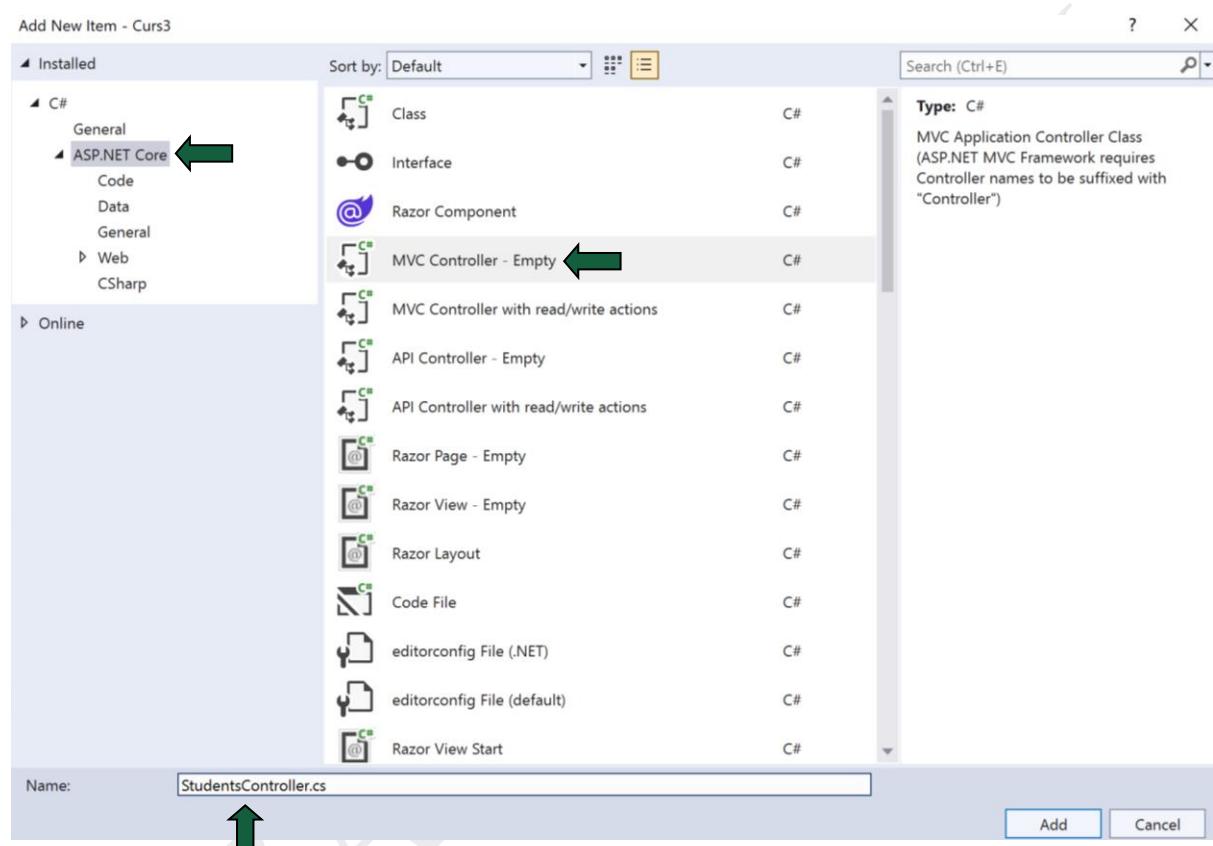
In fereastra aparuta se selecteaza **MVC Controller – Empty**:



### Pas 3:

Se selecteaza din meniul aflat in partea stanga **ASP.NET Core**, dupa care optiunea **MVC Controller – Empty**.

Se adauga numele Controller-ului, nume care trebuie sa aiba sufixul Controller. De exemplu: **StudentsController**.



## Controller-ul implicit – HomeController

In Folderul Controller se observa existenta unui Controller implicit. Acesta este creat automat in momentul in care se creeaza un nou proiect.

### Observatie

Numele unui Controller trebuie sa contine sufixul Controller (ex: **HomeController**).

Controller-ul numit **HomeController** este de fapt o clasa care contine mai multe metode publice (Actions).

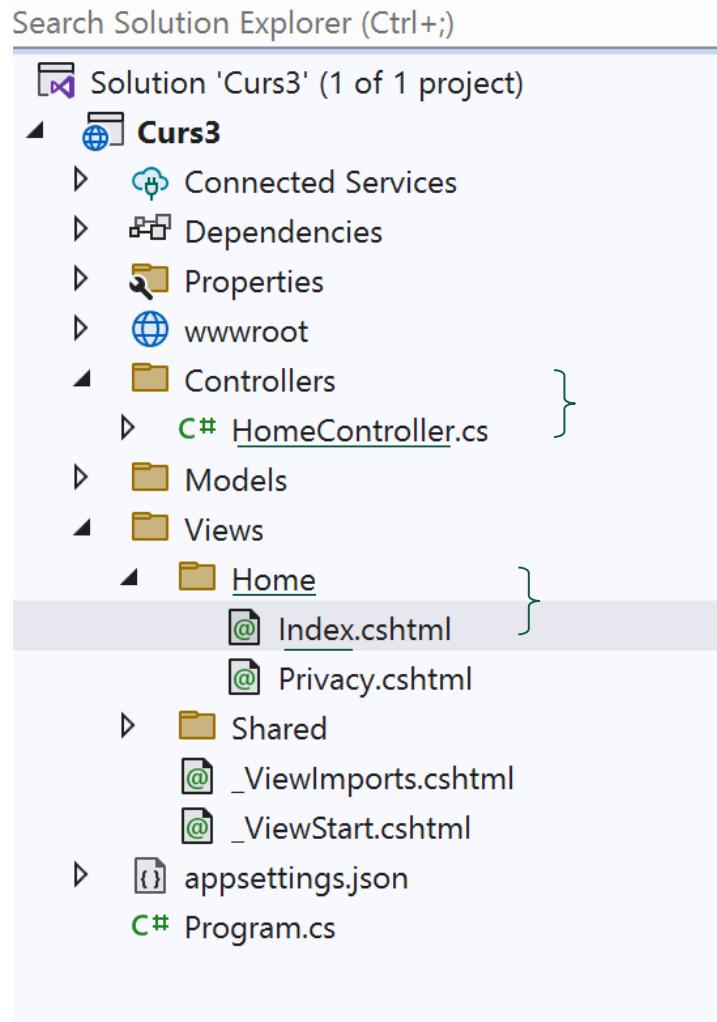
```

Index.cshtml HomeController.cs X Program.cs
Curs3 Curs3.Controllers.HomeController

1  using Curs3.Models;
2  using Microsoft.AspNetCore.Mvc;
3  using System.Diagnostics;
4
5  namespace Curs3.Controllers
6  {
7      public class HomeController : Controller
8      {
9          private readonly ILogger<HomeController> _logger;
10
11         public HomeController(ILogger<HomeController> logger)
12         {
13             _logger = logger;
14         }
15
16         public IActionResult Index()
17         {
18             return View();
19         }
20     }
}

```

Se poate observa metoda **Index()** care returneaza un View. Asadar, metoda Index o sa aiba propriul View, numit exact ca metoda → **Index.cshtml**, aflat in folderul View → Folderul Home (deoarece acesta este numele Controller-ului) (**VEZI imaginea de mai jos**)



## Index.cshtml

```

Index.cshtml  HomeController.cs  Program.cs
@{
    ViewData["Title"] = "Home Page";
}
<div class="text-center">
    <h1 class="display-4">Welcome</h1>
    <p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web
    </div>

```

View-ul anterior, numit **Index.cshtml**, contine doua tipuri de cod:

- O expresie Razor care contine cod C# – un obiect de tip dictionar **ViewData** (vom studia in cursul destinat View-urilor). Codul C# este inclus folosind simbolul @
- Elemente de html pentru interfata (UI)

## Actions

### Structura unei metode

```
public class StudentsController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
}
```

In Controller-ul **StudentsController** se observa metoda Index() de tipul **IActionResult**. Aceasta este definita ca fiind **public** pentru a putea fi accesata de framework.

De asemenea, metodele *trebuie sa fie publice*. Ele nu pot fi private sau protected. Actiunile *nu pot fi supraincarcate* si *nu pot fi statice*.

Tipul returnat **IActionResult** este o interfata, reprezentand raspunsul actiunii trimis de catre Controller la browser, fiind cel mai frecvent tip utilizat deoarece intotdeauna se va apela un View pentru a afisa informatiile catre utilizatorul final.

Metoda **View()** din interiorul actiunii este definita in clasa abstracta de baza **Controller** si este de tipul **ViewResult : ActionResult**.

Deoarece clasa **ActionResult** implementeaza interfata **IActionResult** este suficient ca tipul de return al metodei sa fie **IActionResult**.

## Raspunsul actiunilor – **ActionResult**

Framework-ul MVC include diverse tipuri de rezultat care pot fi returnate prin intermediul **ActionResult**. Aceste clase de rezultat pot fi tipuri de date diferite: html, fisiere, string-uri, json, obiecte, etc.

Posibilele tipuri de date returnate pentru **ActionResult** sunt:

- **ViewResult** – aceasta clasa returneaza continut HTML
- **EmptyResult** – aceasta clasa returneaza un raspuns gol – pagina returnata nu are niciun continut (ex: returneaza status code 200 -> adica request-ul a fost executat correct, dar raspunsul este gol)
- **ContentResult** - poate fi folosit pentru a returna text
- **FileContentResult / FileStreamResult** - reprezinta continutul unui fisier (folosit pentru descarcarea fisierelor)
- **JsonResult** – reprezinta un JSON care poate fi cerut prin AJAX sau alte metode
- **RedirectResult** – reprezinta redirectionarea catre un nou URL
- **RedirectToRouteResult** – reprezinta redirectionarea catre o alta actiune in acelasi Controller sau in alt Controller
- **PartialViewResult** – returneaza HTML-ul dintr-un partial

- **StatusCodeResult** – returneaza un raspuns de tip: BadRequest (400), Unauthorized (401), Forbidden (403), NotFound (404).

**ActionResult** este clasa de baza a tuturor claselor enumerate mai sus. ActionResult implementeaza interfata **IActionResult**. Deci, indiferent de raspunsul folosit, actiunea poate sa aiba tipul de raspuns **IActionResult**.

Pentru a returna tipurile de date mentionate mai sus clasa de baza Controller are urmatoarele metode implementate:

- ViewResult -> View()
- ContentResult -> Content() – primeste ca parametru un string care va fi afisat in browser
- FileContentResult/FilePathResult/FileStreamResult -> File()
- JsonResult -> Json() – primeste ca parametru orice tip de date si va returna un raspuns sub forma JSON (se va serializa parametrul primit sub forma unui string JSON)
- RedirectResult -> Redirect() – primeste ca parametru un URL (in format string) si va redirectiona browser-ul catre acel URL
- RedirectToRouteResult -> RedirectToRoute() – primeste ca parametru un **nume de ruta** (care este definita in fisierul Program.cs) si va redirectiona browser-ul catre acea ruta
- PartialViewResult -> PartialView() – returneaza continutul unui partial

**Avantajul** utilizarii ca tip de date de return **IActionResult** este acela ca se poate utiliza oricare dintre tipurile de raspuns enumerate mai sus fara a schimba tipul de date al actiunii.

De exemplu, pentru a descarca un fisier prin intermediul unei rute putem folosi urmatoarea sevență de cod:

```
public IActionResult Download()
{
    byte[] fileBytes =
System.IO.File.ReadAllBytes(@"c:\folder\myfile.ext");
    string downloadName = "myfile.ext";
    return File(fileBytes,
System.Net.Mime.MediaTypeNames.Application.Octet, downloadName);
}
```

**Pasul 1:** se citeste fisierul ca sevență de bytes de la o cale cunoscută

**Pasul 2:** setam un nume de fisier pentru fisierul care va fi descarcat – downloadName

**Pasul 3:** returnam metoda File care primește 3 parametri:

- Array-ul de bytes care stochează continutul fisierului
- MediaType-ul (MediaType) fisierului descărcat (ex: pentru fisiere **.mp3 -> audio/mpeg**; pentru **imagini de tip JPEG -> image/jpeg**; pentru **imagini de tip PNG -> image/png**) pentru encodarea corecta a fisierului salvat
- Numele fisierului care va fi salvat pe client

## Parametrii unei actiuni

Fiecare actiune poate sa primeasca parametrii unei rute in cadrul semnaturii acesteia. Parametrii rutei pot fi de orice tip (string, int, float sau chiar de tipul clasei unui Model).

```
public IActionResult Show(Student student)
{
    // obiectul student de tipul Modelului Student va
    // contine informatiile unui student din baza de date
}
```

In exemplul anterior, pentru ruta “/Students>Show/2” obiectul student va fi instantiat in mod automat cu valorile studentului cu ID-ul 2 din baza de date, prin intermediul modelului. Acest lucru va fi detaliat in cursul referitor la Model.

### **!OBSERVATIE:**

Numele parametrilor definiti in semnatura Actiunii **trebuie sa fie acelasi** cu numele parametrilor definiti in **Program.cs**. Diferenta dintre numele parametrilor va conduce catre nerezolvarea acestora (vor avea valoarea null).

## Selectori

Framework-ul ASP.NET Core MVC ofera posibilitatea adaugarii unor atribute actiunilor, pentru a ajuta sistemul de rutare sa aleaga actiunea corecta in momentul procesarii unui request. Aceste atribute se numesc **Selectori** si sunt:

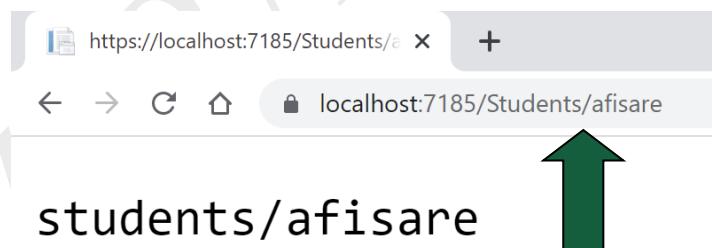
- ActionName
- NonAction
- ActionVerbs

## ActionName

Atributul **ActionName** ofera posibilitatea de a aloca un nume unei actiuni, care este diferit de numele acesteia. De exemplu, daca avem o metoda numita **Index** in Controller-ul **StudentsController** putem redenumi aceasta actiune prin intermediul atributului ActionName astfel:

```
[ActionName("afisare")]
public IActionResult Index()
{
    return Content("students/afisare");
}
```

Inainte de adaugarea atributului ActionName pagina se putea accesa prin URL-ul: **/Students/Index**. Dupa adaugarea atributului ActionName cu valoarea “afisare” pagina poate fi accesata prin intermediul URL-ului **/Students/afisare**. Astfel, acest atribut ne ofera posibilitatea rescrierii numelui actiunii. Acest lucru se intampla fara a aduce modificari fisierului Program.cs.



## NonAction

Atributul **NonAction** indica faptul ca o metoda a unui Controller nu este o actiune. Acest atribut se foloseste in momentul in care dorim ca o metoda publica a unui Controller sa nu poata fi accesata prin intermediul unei rute.

### Exemplu:

```
[NonAction]
public Student GetStudent(int id)
{
    return ...;
}
```

Aceasta metoda nu poate fi accesata prin intermediul unei rute, desi este publica. In schimb, ea poate fi accesata din celelalte metode ale aceluiasi Controller sau ale unui alt Controller.



## ActionVerbs

Atributul **ActionVerbs** este folosit in momentul in care se doreste accesarea unei actiuni in functie de **verbul HTTP**. De exemplu, se pot defini doua actiuni cu acelasi nume, insa care raspund la un verb HTTP diferit si au parametrii diferiti.

**Verbele HTTP** acceptate sunt urmatoarele: **GET, POST, PUT, PATCH, HEAD, OPTIONS** si **DELETE**.

### ⚠️ OBSERVATIE:

In cazul in care atributul ActionVerbs este omis, verbul default folosit este **GET**.

Acstea verbe sunt folosite in urmatoarele contexte:

- **GET**: este folosit in accesarea unei resurse (cererea unei pagini de la server)
- **POST** – este folosit in crearea unei resurse sau trimitera datelor la server prin intermediul unui formular
- **PUT/PATCH** – verbul este folosit pentru modificarea (totala sau parțială) a unei resurse. De exemplu: cand se editeaza o intrare deja existenta in baza de date, se foloseste unul dintre aceste verbe
- **DELETE** – verb folosit pentru stergerea unei resurse
- **HEAD** – este identic cu GET, dar returneaza doar antetele pentru raspuns, nu si continutul raspunsului. De obicei se foloseste pentru a verifica daca exista o resursa sau daca poate fi accesata
- **OPTIONS** – returneaza metodele HTTP acceptate de server pentru o adresa URL specificata

#### **!OBSERVATIE:**

In ASP.NET Core MVC pentru editare (PUT) si stergere (DELETE) se foloseste tot POST → [HttpPost].

## Exemplu definire rute

Exemplu de definire a rutelor folosind **verbele HTTP** corespunzatoare:

```
public class StudentsController : Controller
{
    // GET: lista tuturor studentilor
    public IActionResult Index()
    {
        return View();
    }

    // GET: vizualizarea unui student
    public IActionResult Show(int id)
    {
        return View();
    }
}
```

Acste două actiuni, **Index()** și **Show()** afisează informații despre studenți. **Index** va afisa **lista tuturor studentilor**, iar **Show** va afisa **informații despre un singur student** în funcție de ID-ul primit ca parametru. Deoarece aceste pagini afisează informații și nu trimit nimic la server, vom folosi verbul **GET**. Paginile care au verbul GET se pot accesa direct prin intermediul URL-ului aferent acestora.

```
// GET: se afiseaza formularul de creare a unui student
public IActionResult New()
{
    return View();
}
```

Metoda **New()** care are verbul HTTP GET va afisa prin intermediul view-ului un formular prin care introducem datele aferente unui student. Pentru a trimite datele către server, formularul trebuie să definească metoda prin care trimite datele (adică verbul HTTP → <form action="/students/new" method="post">)

```
[HttpPost]
public IActionResult New(Student student)
{
    // cod creare student
    // dupa crearea studentului, se preia ID-ul nou inserat din
baza de date
    // se redirectioneaza browser-ul catre studentul nou creat

    return Redirect("/students/" + id);
}
```

Datele introduse in formular vor fi trimise catre server prin intermediul metodei POST. Astfel, putem sa definim o ruta cu acelasi nume, dar cu verbul POST [HttpPost]. Aceasta ruta necesita un parametru prin care o sa primeasca datele din formular.

Deoarece metoda are acelasi nume atat in momentul afisarii formularului, cat si in momentul trimiterii datelor catre server, este necesar un tip de date diferit pentru parametrii acesteia, dar si un verb Http diferit.

```
// GET: se doreste editarea unui student
public IActionResult Edit(int ID)
{
    return View();
}

// POST: se trimit modificarile la server si se stocheaza
[HttpPost]
public IActionResult Edit(Student ID)
{
    // cod modificare date student
    // se redirectioneaza browser-ul catre studentul editat
    return Redirect("/Students/Edit" + ID);
    //return RedirectToAction("students_show", new { id = ID
});
```

Aceasta metoda modifica datele studentului si primeste datele prin intermediul verbului HTTP POST. Metodele care creeaza, modifica sau sterg date nu au de obicei un View. Dupa finalizarea procesarii datelor, acestea redirectioneaza utilizatorul la o pagina aferenta actiunii. **De exemplu:** in actiunea de mai sus redirectionam la pagina de afisare a datelor studentului pentru a vedea modificarile efectuate.

```
[HttpPost]
public IActionResult Delete(int id)
{
    // cod stergere student din baza de date
    // redirectionare browser la pagina index a studentilor
    return RedirectToAction("students_index");
}
} // se inchide clasa StudentsController
```

## Redirect in cadrul metodelor

### Redirect

Metoda Redirect se foloseste in momentul in care se doreste realizarea unui redirect temporar (HTTP 302). Primeste ca argument un string, reprezentand URL-ul pe care trebuie sa il acceseze.

Ex: `Redirect("/Students/Edit" + ID);`  
`Redirect("/Home/Index");`

### RedirectToRoute

Metoda RedirectToRoute realizeaza un redirect temporar. Primeste ca argument denumirea rutei, denumire existenta in fisierul Program.cs in momentul configurarii sistemului de rutare.

Ex: `return RedirectToRoute("Nume_Ruta");`

De asemenea, metoda RedirectToRoute mai poate fi utilizata astfel:

```
return RedirectToRoute(new { controller = "Home", action = "Index"});
```

## RedirectToAction

Metoda RedirectToAction se utilizeaza in momentul in care se doreste redirect temporar catre o metoda din acelasi Controller.

Ex: `return RedirectToAction("Edit"); // metoda din acelasi Controller`

Se poate redirectiona si catre o metoda dintr-un alt Controller astfel:

```
return RedirectToAction("Nume_Actiune", "Nume_Controller"); →  
→ return RedirectToAction("Index", "Students");  
// redirect catre metoda Index din Controller-ul Students
```

## RedirectPermanent/ RedirectToRoutePermanent/ RedirectToActionPermanent

Acste metode se utilizeaza la fel ca in exemplele anterioare, singura diferenta fiind starea redirect-ului. In acest caz se realizeaza un redirect permanent (HTTP 301). Raspunsul o sa fie stocat in memoria cache a browser-ului, iar serverul nu o sa mai fie interogat pentru accesarile ulterioare.

## Returnare HTTP Status Code

Pentru a returna un status al request-ului HTTP, se poate proceda astfel:

```
public StatusCodeResult BadRequest()
{
    return StatusCode(StatusCodes.Status400BadRequest);
}

public StatusCodeResult Unauthorized()
{
    return StatusCode(StatusCodes.Status401Unauthorized);
}

public IActionResult Forbidden()
{
    return StatusCode(StatusCodes.Status403Forbidden);
}

public IActionResult NotFound()
{
    return StatusCode(StatusCodes.Status404NotFound);
    //sau return NotFound();
}
```

Ca tip de returnare, la fel ca in cazul celorlalte tipuri, se poate utiliza direct IActionResult.

# Dezvoltarea Aplicatiilor Web utilizand ASP.NET Core MVC

## Laborator 4

---

### EXERCITII:

#### Exercitiul 1:

Sa se creeze un nou proiect si sa se adauge Controller-ul **ArticlesController** in care se vor implementa urmatoarele (**VEZI** Curs 4 – capitolul Controller):

1. Sa se adauge un Model numit **Article** care sa contine **Id**, **Title**, **Content** si **Date** astfel: Models -> click dreapta -> Add -> Class -> Adaugam o clasa Article.cs

```
public class Article
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public DateTime Date { get; set; }
}
```

2. Sa se adauge in Controller o metoda **NonAction**, numita **GetArticles()** (**VEZI** Curs 4 – in sectiunea **Selectori**) care va returna un array de obiecte de tip Article, array pe care o sa il folosim pentru afisare (procedam astfel deoarece in acest laborator nu o sa folosim baza de date, iar in acest mod ne cream articole pe care le putem prelucra).

(\*) Cum se poate implementa o metoda NonAction? Ce reprezinta o metoda NonAction?

```
public Article[] GetArticles()
{
    // Se instantiaza un array de articole
    Article[] articles = new Article[3];

    // Se creeaza articolele
    for (int i = 0; i < 3; i++)
    {
        Article article = new Article();
        article.Id = i;

        article.Title = "Articol " + (i + 1).ToString();
        article.Content = "Continut articol " + (i + 1).ToString();
        article.Date = DateTime.Now;

        // Se adauga articolul in array
        articles[i] = article;
    }
    return articles;
}
```

3. Sa se adauge toate metodele pentru operatiile de tip C.R.U.D.

- **Index** – pentru listarea tuturor articolelor
- **Show** – pentru vizualizarea unui articol in functie de Id
- **New** – pentru crearea unui nou articol
- **Edit** – pentru editarea unui articol existent
- **Delete** – pentru stergerea unui articol

De asemenea, se vor adauga verbele HTTP potrivite pentru fiecare metoda in parte (**VEZI** Curs 4 – sectiunea Selectorii).

#### 4. Index

- Sa se creeze metoda Index in Controller-ul ArticlesController
- Sa se creeze un **View** numit **Index**

In folderul View → se adauga un folder nou corespunzator Controller-ului Articles astfel → Click dreapta pe folderul Views → Add → New Folder → Folderul o sa se numeasca Articles. In acest folder se creeaza un fisier cs.html asociat metodei Index → Click dreapta pe folderul Articles → Add → View → Se selecteaza optiunea Razor View Empty → Add → Se selecteaza ASP.NET Core si Razor View – Empty → Se modifica denumirea in Index → Add

- In continuare se utilizeaza array-ul de articole creat, astfel incat sa preluam in metoda Index toate articolele, dupa care sa le trimitem catre View-ul asociat pentru afisare catre utilizatorul final

#### Metoda Index din ArticlesController:

```
public IActionResult Index()
{
    Article[] articles = GetArticles();

    // Se adauga array-ul de articole in View
    ViewBag.Articles = articles;

    return View();
}
```

#### Index.cshtml – in View-ul Articles

Sa se implementeze afisarea articolelor. De asemenea, pentru fiecare articol trebuie sa existe posibilitatea afisarii si editarii acestuia.

5. Modificati ruta **Default** din Program.cs, astfel incat la rularea proiectului sa ne redirectioneze de fiecare data catre pagina de listare a tuturor articolelor.

## 6. Show

- In metoda Show din ArticlesController afisati detaliile unui articol, in functie de id-ul acestuia. In cazul in care articolul nu este gasit, afisati un View de eroare cu explicatiile aferente (mesajul de eroare aruncat de exceptie si un mesaj – “Articolul cautat nu poate fi gasit”).
- Sa se creeze un View numit Show, in care sa se implementeze afisarea unui singur articol. View-ul va contine si un link catre pagina de afisare a tuturor articolelor.
- Configurati o ruta pentru ruta existenta **/Articles>Show/{id}**, care dupa cum se observa se poate accesa prin intermediul rutei Default, astfel incat sa se acceseze prin ruta **/articole/show/{id}**. Ruta o sa se numeasca **ArticlesShow**

### ArticlesController → Show

```
// Afisarea unui singur articol
public IActionResult Show(int? id)
{
    Article[] articles = GetArticles();

    try
    {
        ViewBag.Article = articles[(int)id];
        return View();
    }

    catch (Exception e)
    {
        ViewBag.ErrorMessage = e.Message;
        return View("Error");
    }
}
```



## View → Show.cshtml

```
@{
    ViewBag.PageName = "Show";
}

<h2>@ViewBag.PageName</h2>

<hr />

<h1>@ViewBag.Article.Title</h1>
<p>@ViewBag.Article.Content</p>
<small>@ViewBag.Article.Date</small>

<hr />
<br />

<a href="/Articles/Index">Afisare articole</a>
```

## View → Error.cshtml

```
@{
    ViewBag.Msg = "Articolul cautat nu poate fi gasit!";
}

<h2>@ViewBag.Msg</h2>
<br />
<p>Error message: @ViewBag.ErrorMessage</p>
```

## Index.cshtml → ruta configurata

```
<a href="/articole/show/@article.Id">Afisare articol</a>
// Ruta configurata
```

## 7. New

- In Controller o sa existe doua metode numite **New**.  
**Prima metoda** este de tip [HttpGet] si se utilizeaza pentru afisarea formularului prin intermediul caruia se completeaza datele unui articol.  
**A doua metoda** este utilizata pentru trimitera datelor corespunzatoare articolului catre server, pentru adaugarea noului articol in baza de date. Pentru trimitera datelor intotdeauna se foloseste verbul [HttpPost].
  
- O sa existe, de asemenea, doua View-uri.  
**Un View** pentru afisarea formularului de creare a unui articol.  
**Al doilea View** (caruia ii dam un nume diferit – de exemplu: NewPostMethod) folosit in momentul in care datele pentru creare vor fi trimise catre server. In acest caz, al doilea View o sa afiseze doar un mesaj deoarece nu avem inca acces la o baza de date.
  
- Link-ul catre pagina de creare a unui articol o sa fie adaugat in Index.

### ArticlesController → New ([HttpGet])

```
// GET: Afisarea formularului de creare a unui articol
[HttpGet]
public IActionResult New()
{
    return View();
}
```

## ArticlesController → New ([HttpPost])

```
// POST: Trimiterea datelor despre articol catre server
pentru adaugare in baza de date

[HttpPost]
public IActionResult New(Article article)
{
    // ... cod creare articol ...

    return View("NewPostMethod");
}
```

## View → New.cshtml

```
<h2>Afisare formular de adaugare articol</h2>
<form method="post" action="/Articles/New">
    <button type="submit">Adauga articol</button>
</form>
```

## View → NewPostMethod.cshtml

```
@{
    ViewBag.New = "Articolul a fost adaugat cu succes!";
}

<h2>
    @ViewBag.New
</h2>
```

## 8. Edit

- In Controller o sa existe doua metode numite **Edit**.  
**Prima metoda** este de tip [HttpGet] si se utilizeaza pentru afisarea formularului care o sa contine datele unui articol existent in baza de date. Formularul se afiseaza cu scopul modificarii campurilor (o parte din ele sau chiar toate).  
**A doua metoda** este utilizata pentru trimitera datelor corespunzatoare articolului catre server, pentru editarea articolului in baza de date. Pentru trimitera datelor intotdeauna se foloseste verbul [HttpPost]. In cazul API-urilor se utilizeaza [HttpPut], dar in ASP.NET Core se poate utiliza [HttpPost] atat pentru New, cat si pentru Edit si Delete). A doua metoda poate returna un View, la fel ca in exemplul anterior sau se poate redirectiona catre pagina principala a aplicatiei, pagina in care se afiseaza lista tuturor articolelor.
  
- Si in acest caz o sa existe doua View-uri.  
**Un View** pentru afisarea articolului din baza de date, articol care urmeaza sa fie editat.  
**Al doilea View** (caruia ii dam un nume diferit – de exemplu: EditMethod) este folosit in momentul in care datele pentru editare vor fi trimise catre server. In acest caz, al doilea View o sa afiseze doar un mesaj deoarece nu avem inca acces la o baza de date.

### ArticlesController → Edit ([HttpGet])

```
// GET: Afisarea datelor unui articol pentru editare

[HttpGet]
public IActionResult Edit(int? id)
{
    ViewBag.Id = id;
    return View();
}
```

## ArticlesController → Edit ([HttpPost])

```
// POST: Trimiteara modificarilor facute catre server
pentru stocare in baza de date

[HttpPost]
public IActionResult Edit(Article article)
{
    // ... cod adaugare articol editat in baza de date
    //return Redirect("/Articles/Index");
    return View("EditMethod");
}
```

## View → Edit.cshtml

```
<br />
<p>Afisare formular de editare articol - in acest formular
de preiau datele curente ale articolului</p>
<form method="post" action="/Articles/Edit/@ ViewBag.Id">
    <button type="submit">Editeaza articol</button>
</form>
```

## View → EditMethod.cshtml

```
@{
    ViewBag.New = "Articolul a fost editat cu succes!";
}
<h2> @ViewBag.New </h2>
```

## 9. Delete

- In pagina Show inserati formularul corespunzator stergerii intrarii prin **metoda DELETE**, folosind un buton la fel ca in exemplele anterioare. Verbul folosit este [HttpPost]. In cazul implementarii unui API, verbul HTTP o sa fie [HttpDelete]. In ASP.NET Core se poate utiliza [HttpPost] atat pentru New, cat si pentru Edit si Delete.

### ArticlesController → Delete ([HttpPost])

```
// POST Stergere articol din baza de date

[HttpPost]
public IActionResult Delete(int? id)
{
    // ... cod stergere articol din baza de date

    return Content("Articolul a fost sters din baza de
date!");}
```

### View → Show.cshtml

```
<form method="post" action = "/Articles/Delete/@ViewBag.Article.Id">

<button type="submit">Stergere articol</button>

</form>
```

10. Redenumiti metoda Index din Controller in *listare*, folosind selectori (**VEZI** Curs 4 – sectiunea Selectorii). **Ce observati dupa redenumire?** **Ce modificari trebuie realizate?**

# Dezvoltarea Aplicatiilor Web utilizand ASP.NET Core MVC

## Curs 5

---

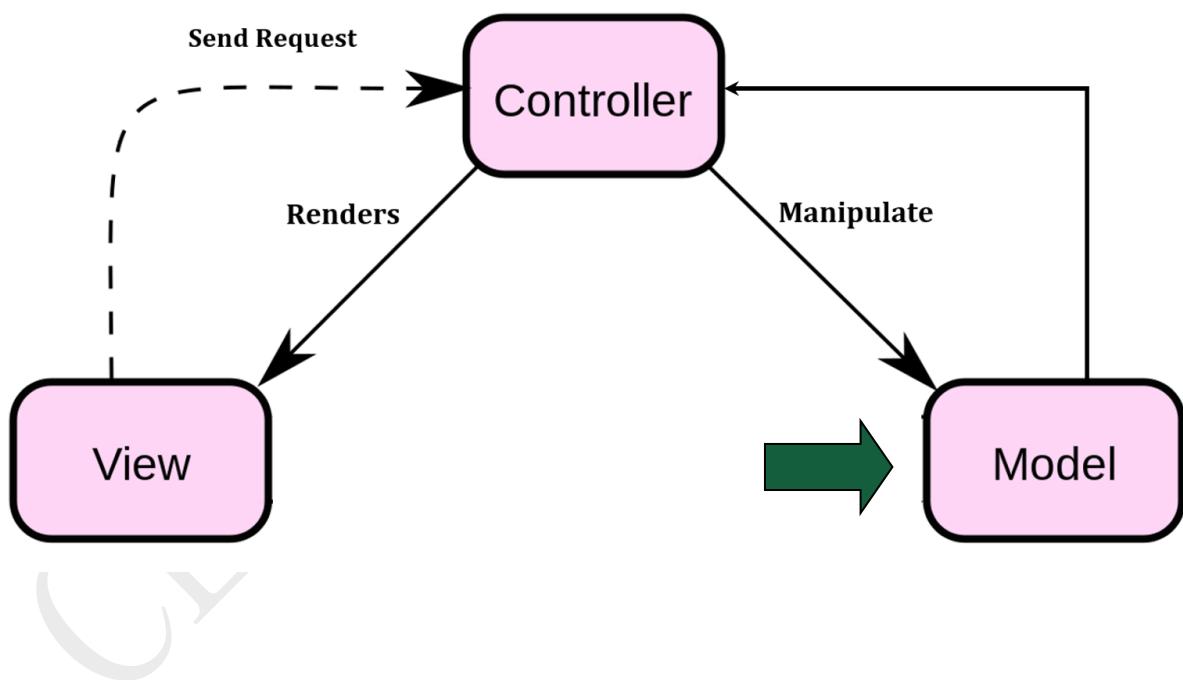
### Cuprins

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 – Migratii .....	11
Ce sunt migratiile? .....	11
Crearea unui proiect utilizand EF si sistemul de migratii.....	12
Crearea proiectului .....	12
Adaugare Entity Framework Core .....	12
Adaugarea Modelului.....	12
Conexiunea cu Baza de Date .....	15
Adaugarea unei baze de date SQL Server .....	19
Crearea migratiilor in baza de date .....	24
C.R.U.D. utilizand Entity Framework .....	26
Index.....	26
Show.....	27
New .....	28
Model Binding .....	29
Edit .....	30
Delete .....	31

## Model (Stratul business – prelucrarea datelor)

### Ce este Modelul

**Modelul** este responsabil cu gestionarea datelor din aplicatie si manipularea acestora. Acesta raspunde cererilor care vin din View prin intermediul Controller-ului, modelul comunicand doar cu Controller-ul. Este cel mai de jos nivel care se ocupa cu **procesarea si manipularea** datelor, reprezentand nucleul aplicatiei, fiind cel care realizeaza legatura cu baza de date. **Modelul** ofera accesul la date prin intermediul atributelor publice ale claselor.



# Entity Framework Core

## Entity Framework

Pentru stocarea datelor in ASP.NET MVC se utilizeaza o tehnologie numita open source numita **Entity Framework (EF)**.

Entity Framework este un **ORM (Object Relational Mapper)** pentru .NET, si anume este o colectie de librarii care coreleaza fiecare clasa dintr-un model cu o baza de date. Scopul utilizarii EF este acela de a permite dezvoltatorilor sa se focuseze pe dezvoltarea propriu-zisa a aplicatiei si nu pe baza de date.

Procesarea datelor se poate realiza si prin metode clasice, de exemplu utilizand ADO.NET, dar EF ofera posibilitatea implementarii facile a operatiilor de tip CRUD (Create, Read, Update, Delete).

De asemenea, in cadrul EF se poate utiliza **LINQ (Language Integrated Query)** ajutand la integrarea oricarui **RDBMS (Relational Database Management System)** → Oracle SQL, SQL Server, etc. Un RDBMS stocheaza date in tabele, pe care ulterior le acceseaza si prelucreaza cu ajutorul unui limbaj **SQL (Structured Query Language)**. Un RDBMS asigura securitatea, integritatea si consistenta datelor.

**LINQ** permite integrarea query-urilor SQL in cadrul codului C#.



## Entity Framework Core

**Entity Framework Core** este versiunea mai light a EF, cross-platform (Linux, Windows) care functioneaza foarte bine impreuna cu ASP.NET Core. Entity Framework Core suporta, la fel ca EF, atat tehnica **database-first** cat si **code-first**.

EF Core contine atat posibilitatea integrarii unui RDBMS (Oracle SQL, Microsoft SQL Server, MySQL), cat si integrarea unor baze de date non-relationale (MongoDB, Redis, CassandraDB).

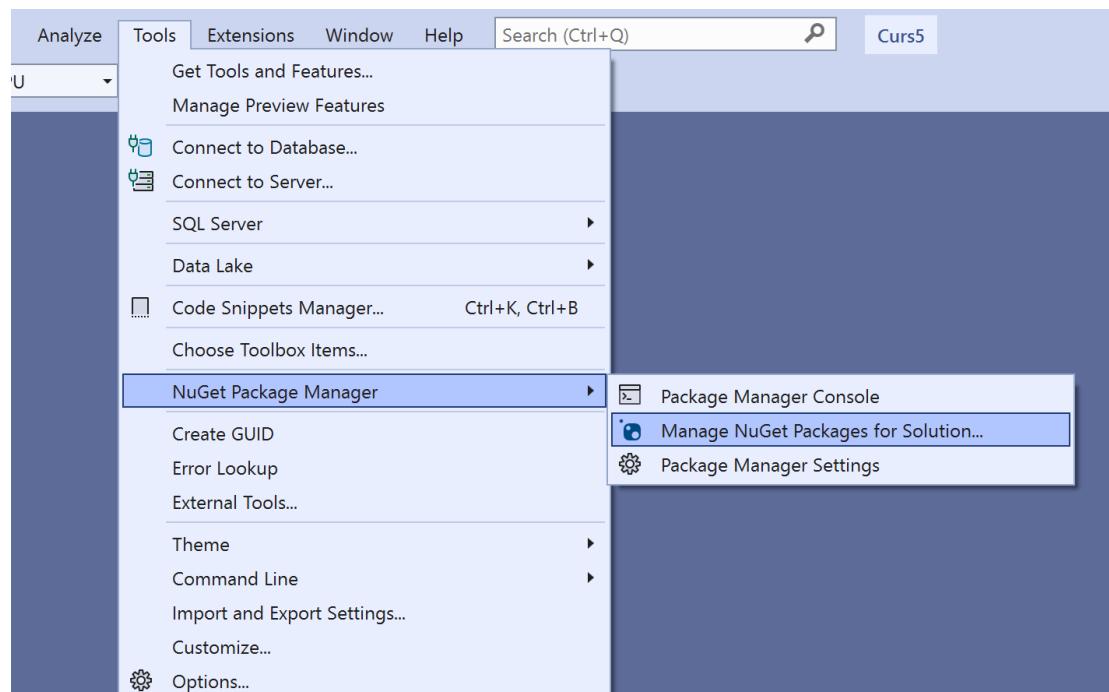
## Instalare Entity Framework Core

Entity Framework este un pachet care poate fi instalat folosind **NuGet** si care suporta tehnica **code-first**. Tehnica code-first ofera posibilitatea dezvoltatorilor de a scrie clase prin intermediul carora baza de date va fi generata automat. Acest lucru duce la o dezvoltare curata si rapida a aplicatiilor cu baze de date.

Pentru instalarea Entity Framework (EF) se procedeaza astfel:

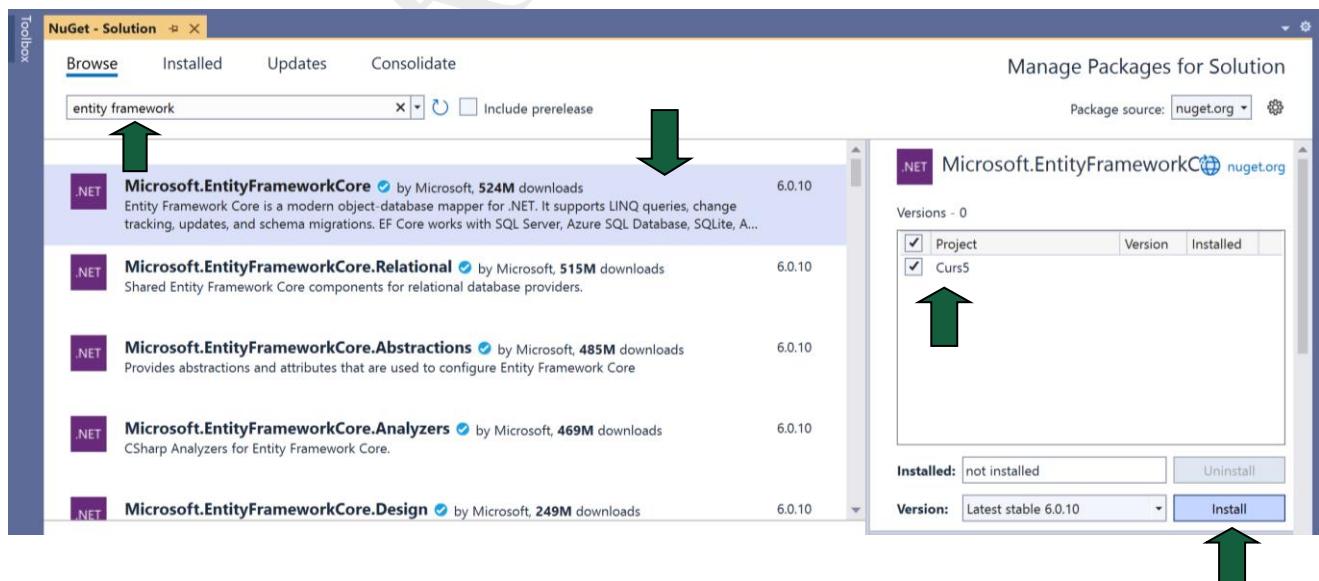
### PASUL 1:

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

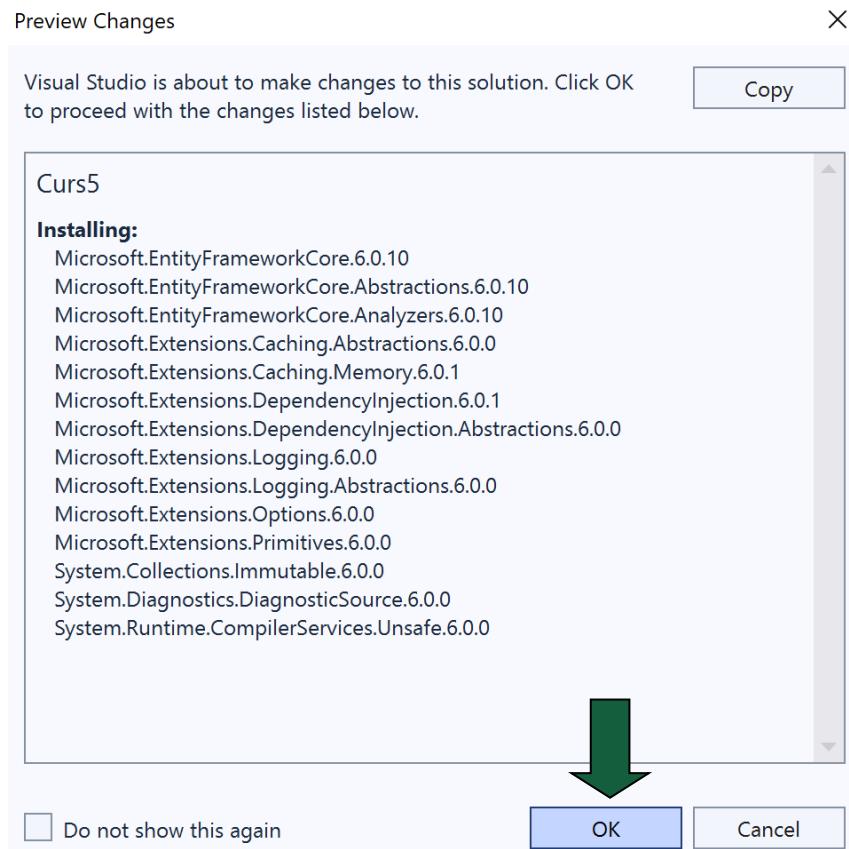


## PASUL 2:

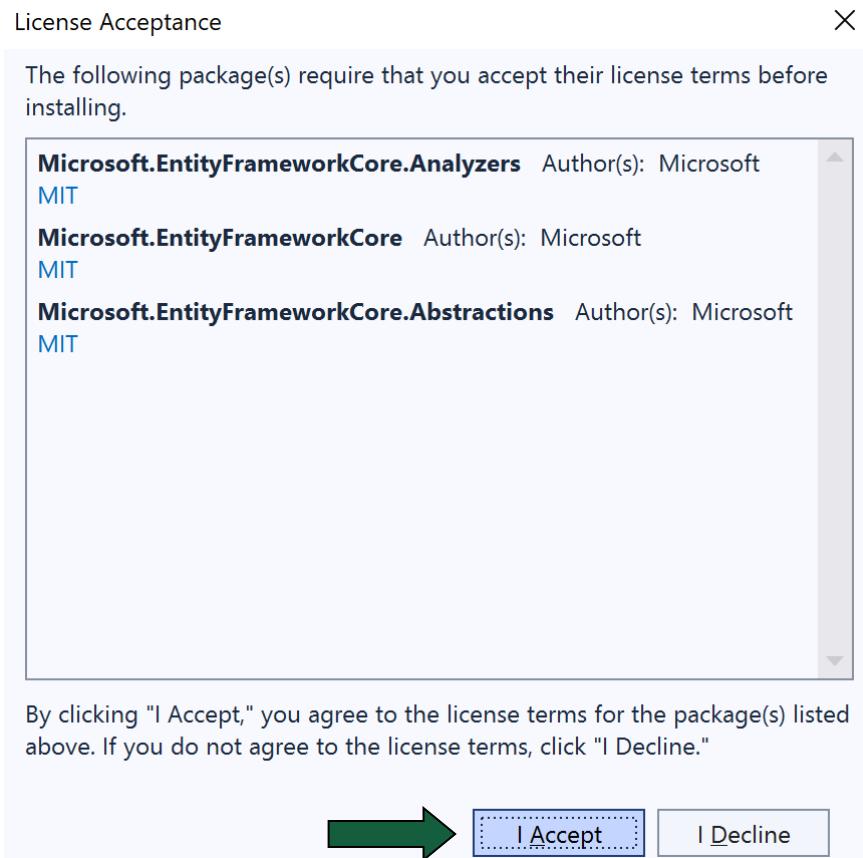
Se acceseaza optiunea **Browse** si se cauta  
**Microsoft.EntityFrameworkCore** dupa cum urmeaza:



## PASUL 3:

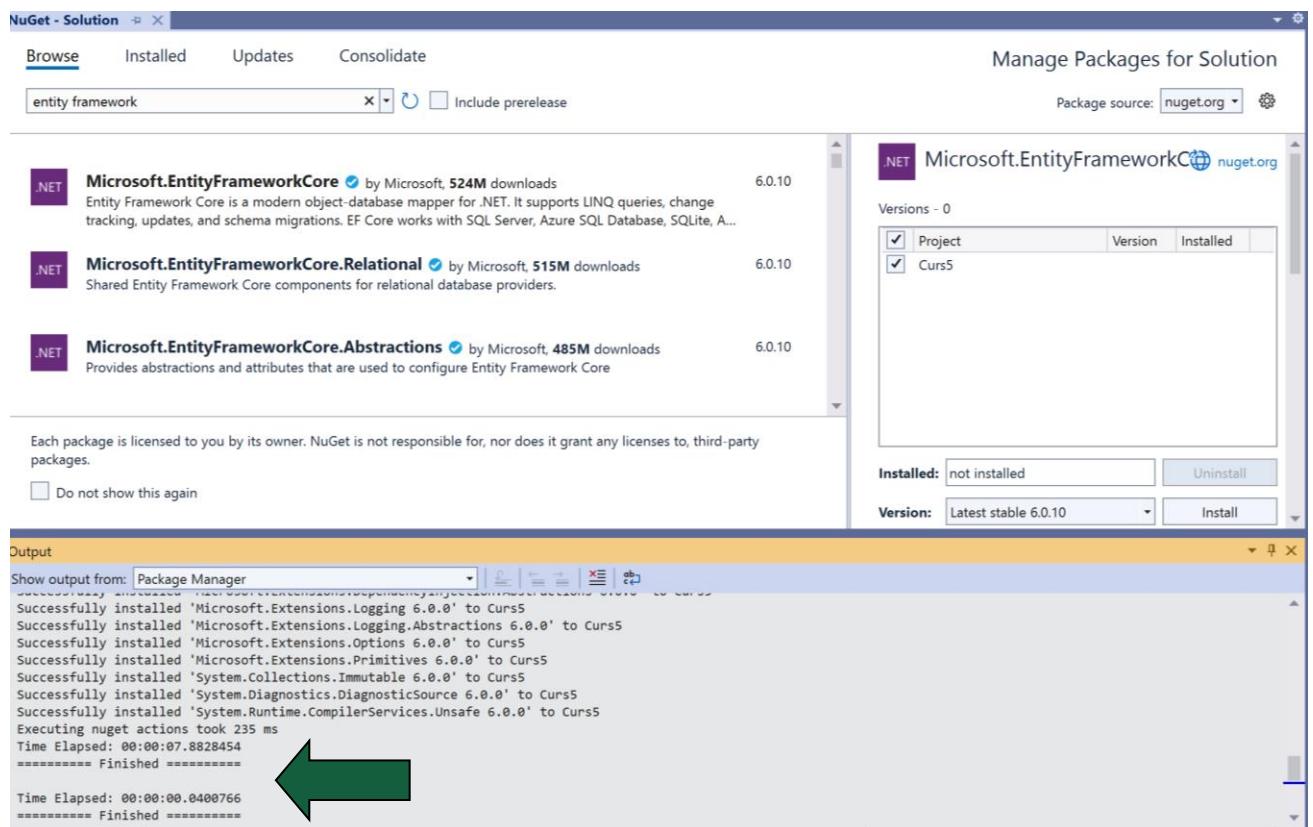


## PASUL 4:

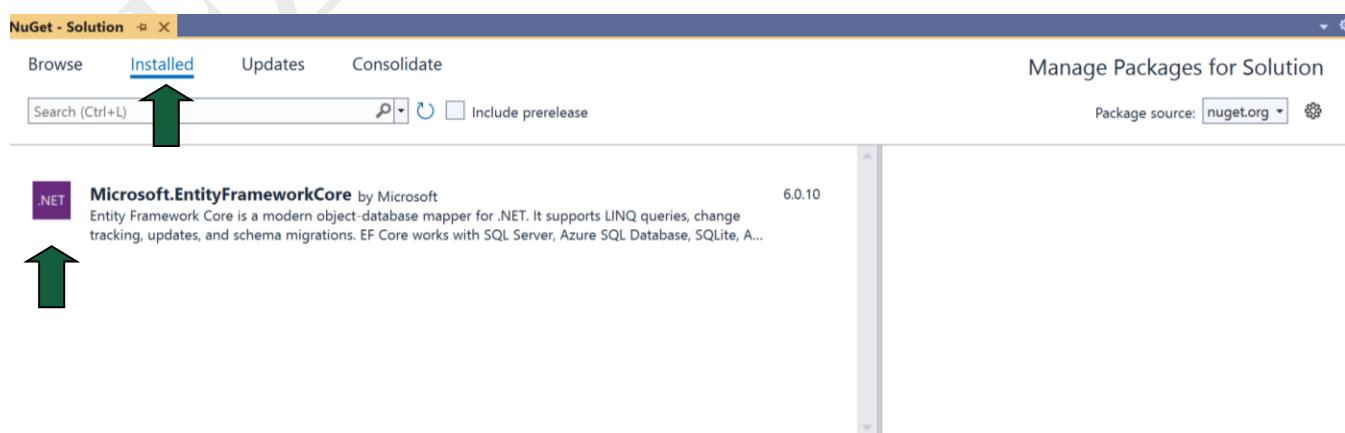


## PASUL 5:

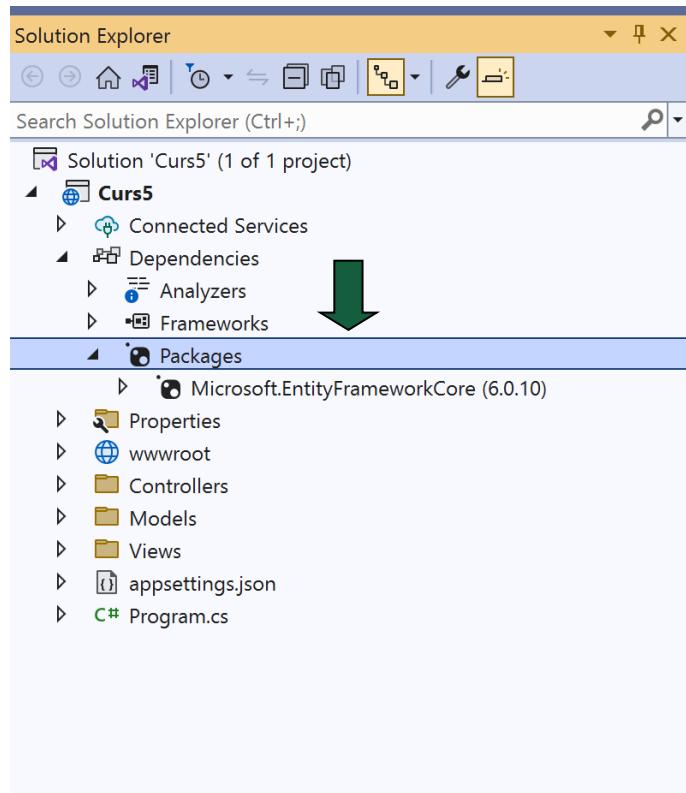
Dupa instalare se poate verifica in consola daca EF a fost adaugat cu succes in cadrul proiectului.



De asemenea, se poate verifica si in sectiunea **Installed**.



Sau chiar in Solution Explorer → Dependencies → Packages



In continuare este necesara includerea pachetului **SqlServer** (pentru baza de date) si **Design** (pachet care contine tool-urile necesare pentru rularea comenzilor de migrare).

Se instaleaza si pachetul Tools care include comenzi precum: Add-Migration, Drop-Database, Get-DbContext, Remove-Migration , etc.

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

Browse    Installed    Updates    Consolidate

entity framework       Include prerelease

Manage Packages for Solution

Package source: nuget.org

.NET	Microsoft.EntityFrameworkCore	by Microsoft, 524M downloads	6.0.10
	<b>Microsoft.EntityFrameworkCore.Relational</b>	Entity Framework Core is a modern object-database mapper for .NET. It supports LINQ queries, change tracking, updates, and schema migrations. EF Core works with SQL Server, Azure SQL Database, SQLite,...	6.0.10
	<b>Microsoft.EntityFrameworkCore.Abstractions</b>	Shared Entity Framework Core components for relational database providers.	6.0.10
	<b>Microsoft.EntityFrameworkCore.Analyzers</b>	Provides abstractions and attributes that are used to configure Entity Framework Core	6.0.10
	<b>Microsoft.EntityFrameworkCore.Design</b>	CSharp Analyzers for Entity Framework Core.	6.0.10
	<b>Microsoft.EntityFrameworkCore.SqlServer</b>	Shared design-time components for Entity Framework Core tools.	6.0.10
	<b>Microsoft.EntityFrameworkCore.Tools</b>	Microsoft SQL Server database provider for Entity Framework Core.	6.0.10

.NET Microsoft.EntityFrameworkCore nuget.org

Versions - 0

Project	Version	Installed
<input checked="" type="checkbox"/> Curs5		

Installed: not installed

Version: Latest stable 6.0.10

Options

Description

Microsoft SQL Server database provider for Entity Framework Core.

Version: 6.0.10  
Author(s): Microsoft

Browse    Installed    Updates 1    Consolidate

entity framework       Include prerelease

Manage Packages for Solution

Package source: nuget.org

.NET	Microsoft.EntityFrameworkCore.Relational	by Microsoft, 515M downloads	6.0.10
	<b>Microsoft.EntityFrameworkCore.Abstractions</b>	Shared Entity Framework Core components for relational database providers.	6.0.10
	<b>Microsoft.EntityFrameworkCore.Analyzers</b>	Provides abstractions and attributes that are used to configure Entity Framework Core	6.0.10
	<b>Microsoft.EntityFrameworkCore.Design</b>	Shared design-time components for Entity Framework Core tools.	6.0.10
	<b>Microsoft.EntityFrameworkCore.SqlServer</b>	Microsoft SQL Server database provider for Entity Framework Core.	6.0.10
	<b>Microsoft.EntityFrameworkCore.Tools</b>	Entity Framework Core Tools for the NuGet Package Manager Console in Visual Studio.	6.0.5
	<b>EntityFramework</b>	Entity Framework 6 (EF6) is a tried and tested object-relational mapper for .NET with many years of feature development and stabilization.	6.4.4

.NET Microsoft.EntityFrameworkCore nuget.org

Versions - 0

Project	Version	Installed
<input checked="" type="checkbox"/> Curs5		

Installed: not installed

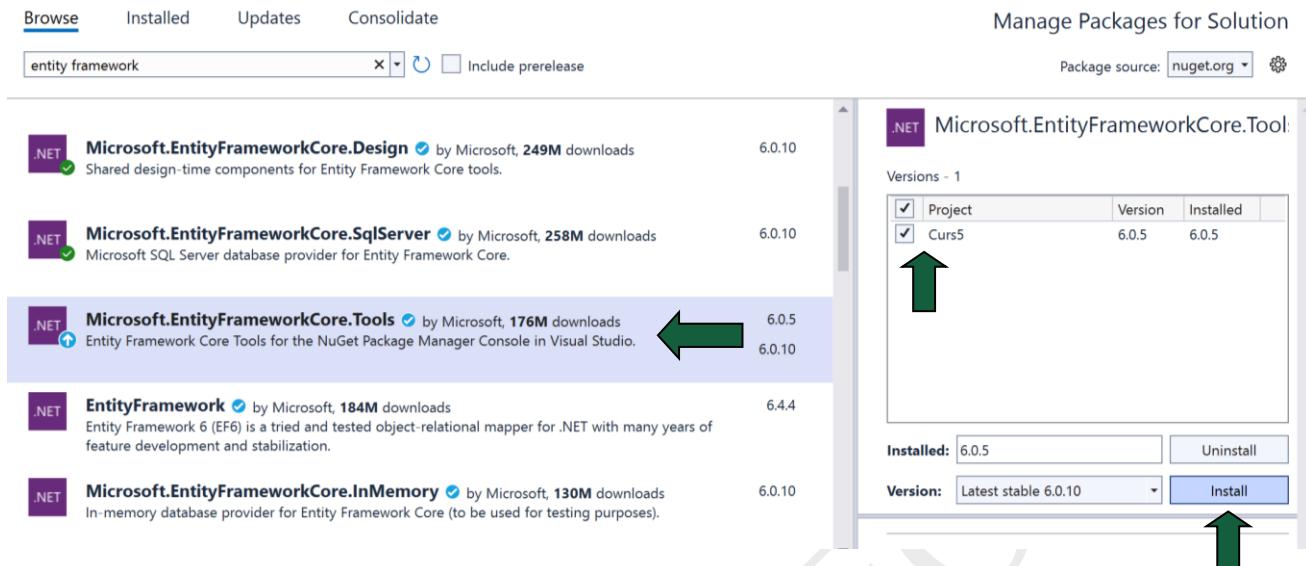
Version: Latest stable 6.0.10

Options

Description

Shared design-time components for Entity Framework Core tools.

Version: 6.0.10  
Author(s): Microsoft



## Entity Framework Core – Migratii

### Ce sunt migratiile?

In timpul dezvoltarii unei aplicatii baza de date se modifica constant, fiind necesare entitati noi, proprietati noi sau chiar eliminarea unor proprietati existente. Pentru a sincroniza aceste modificari cu baza de date existenta, sunt necesare **migratiile**.

In momentul in care apare o modificare in baza de date, Entity Framework Core, prin sistemul de migratii, compara modelul curent cu cel anterior pentru a detecta diferentele dintre cele doua versiuni. Ulterior este generat un fisier, continand codul asociat migratiei.

# Crearea unui proiect utilizand EF si sistemul de migratii

## Crearea proiectului

Se creeaza un nou proiect, procedand la fel ca in cursurile anterioare. Proiectul o sa se numeasca **Curs5**.

## Adaugare Entity Framework Core

In cadrul noului proiect se adauga EF (**VEZI** Sectiunea – Instalare Entity Framework Core – din cadrul cursului curent).

## Adaugarea Modelului

Pentru adaugarea unui model se porneste de la crearea in cadrul acestuia a tuturor entitatilor. Prin **entitate** ne referim la un tabel din baza de date.

**Entitate** = un loc, o actiune, o persoana, etc.

**Exemplu de entitati** dintr-o baza de date care gestioneaza o Universitate → Studenti, Cursuri, Note, MergeLa – tabel asociativ intre Studenti si Cursuri

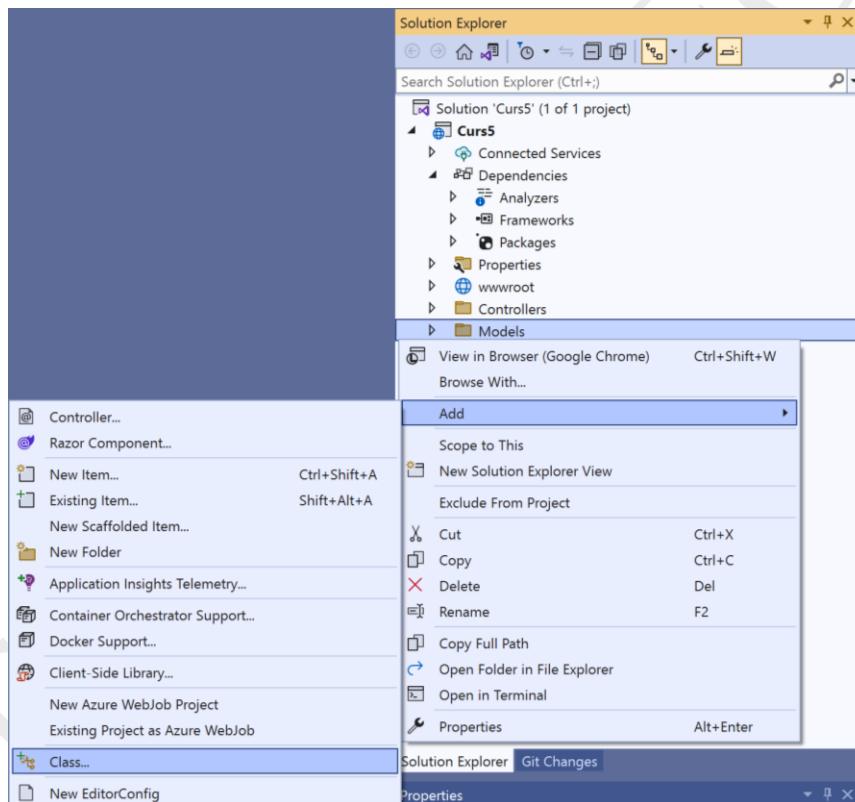
In continuare vom adauga clasa **Student** cu urmatoarele atribute:

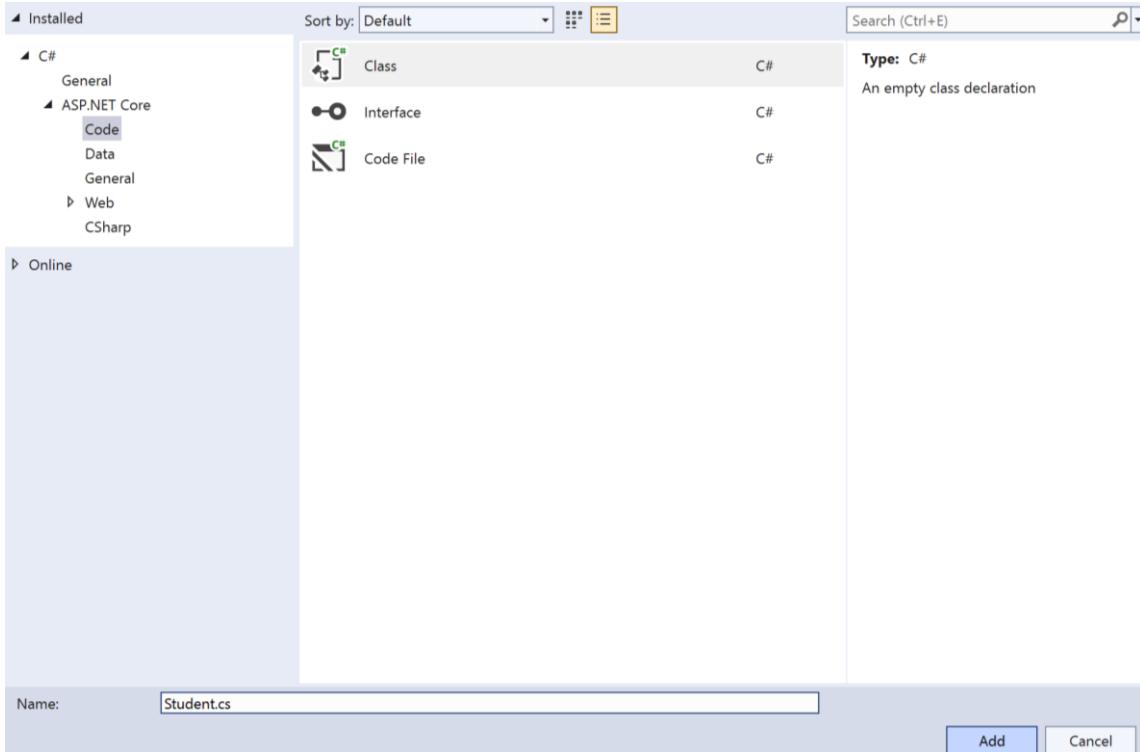
- **StudentID** – de tip int care reprezinta ID-ul studentului
- **Name** – de tip string care reprezinta numele studentului
- **Email** – de tip string care reprezinta o adresa de e-mail a studentului

- **CNP** – de tip string care reprezinta CNP-ul studentului. Aceasta proprietate a fost definita de tip string deoarece spatiul alocat pentru int nu suporta valori de 13 caractere. De asemenea, definit ca string se poate procesa caracter cu caracter pentru calcule ulterioare (ex: extragere data de nastere).

Pentru adaugarea clasei **Student** se parcurg urmatorii pasi:

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





Clasa **Student** din fisierul 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; }
}
```

Aceasta clasa reprezinta tabelul **Student** din baza de date, pe care in varianta clasica il adaugam prin intermediul comenzii **CREATE TABLE**.

Atributele / proprietatile clasei sunt: StudentID, Name, Email, CNP. Acestea reprezinta coloanele din tabelul Student, coloane care sunt create in momentul in care se creeaza si tabelul → utilizand comanda **CREATE TABLE**

## Conexiunea cu Baza de Date

Conexiunea cu baza de date se poate realiza in doua moduri:

- fara dependency injection
- cu dependency injection.

### Varianta 1 – fara Dependency Injection

Pentru a putea adauga layer-ul de conexiune cu baza de date in cadrul unui model, este necesara adaugarea unei noi clase.

Se adauga o clasa, numita 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** mosteneste clasa de baza **DbContext** din Entity Framework. Clasa de baza realizeaza in mod automat conexiunea cu baza de date, crearea tabelului daca acesta nu exista si contine o proprietate **DbSet**, care trebuie sa primeasca tipul modelului (Student in cazul curent) si numele pluralizat al modelului.

**DbSet <Student> Students { get; set; }** – prin intermediul acestei sechente de cod vom avea acces la intrarile din baza de date; se pot interoga si stoca instante de tip Student.

Stringul de conexiune la baza de date este preluat de parametrul **options** din metoda **OnConfiguring** prin intermediul urmatoarei sechente de cod:

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

**UseSqlServer()** → metoda prin intermediul careia se configureaza contextul pentru conectarea la baza de date. Primeste ca argument stringul de conectare la baza de date. **Stringul de conectare la baza de date se obtine urmand sectiunea urmatoare din curs.**

In final se adauga in **Program.cs** urmatoarea sechenta de cod pentru initializarea bazei de date.

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

In cadrul fiecarui Controller trebuie sa se instantieze contextul pentru realizarea conexiunii la baza de date.

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

## Varianta 2 – cu Dependency Injection

Se adauga aceeasi clasa, numita sugestiv **AppDbContext** care in acest caz o sa aiba 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 in acest caz exclusiv in **Program.cs**.

```

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

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

```

In acest caz, stringul de conexiune o sa fie in **Solution Explorer → appsetting.json** adaugand secheta de cod marcata cu **BOLD** si accolada.

```

{
  "ConnectionStrings": {
    "DefaultConnection": {
      "Server=(localdb)\\mssqllocaldb;Database=aspnet-53bc9b9d-9d6a-
      5d4-8429-
      12761773502;Trusted_Connection=True;MultipleActiveResultSets=tr
      ?"
    },
    "Logging": {
      "LogLevel": {
        "Default": "Information",
        "Microsoft.AspNetCore": "Warning"
      }
    },
    "AllowedHosts": "*"
  }
}

```

In final, in cadrul fiecarui Controller se realizeaza conexiunea cu baza de date astfel:

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

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

    ...
}
```

Pentru mentenanta mai facila a codului sursa framework-ul ASP.NET Core foloseste **dependency injection** pentru a insera automat la runtime instante ale unor obiecte necesare. Acest lucru se intampla prin intermediul constructorilor care se afla in Controllere. Atfel, un Controller poate cere o instanta a unui obiect din cadrul framework-ului doar prin specificarea unui parametru si tipul acestuia.

Un astfel de exemplu este conexiunea la baza de date care se cere prin intermediul constructorului specificand un parametru de tip AppDbContext (in cazul nostru particular din curs unde am numit clasa in acest mod). In acest caz framework-ul stie in mod automat sa returneze o instanta a AppDbContext din serviciul aferent bazei de date, configurat in Program.cs.

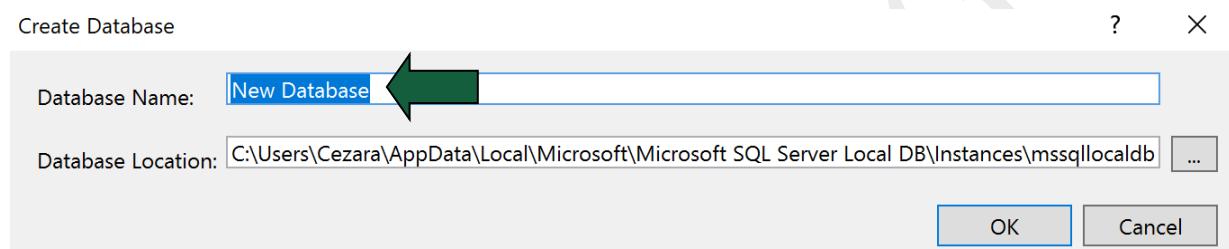


## Adaugarea unei baze de date SQL Server

Pentru adaugarea baze de date se parcurg urmatorii pasi:

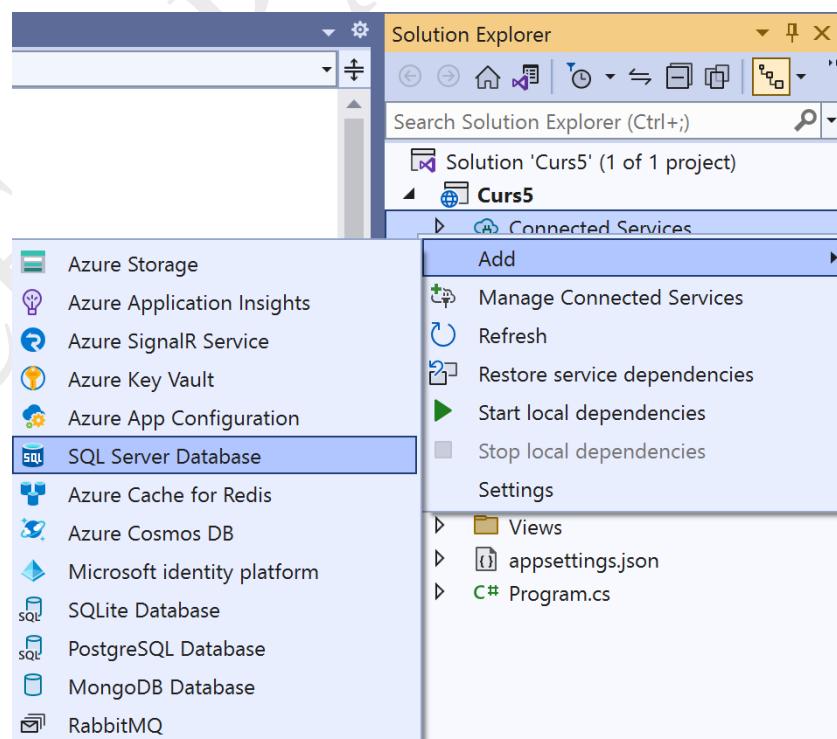
### PASUL 1:

In SQL Server Object Explorer (se afla in meniul View) → se deschide sectiunea (localdb)\MSSQLLOCALDB → in folderul Databases → click dreapta → Add new database → se completeaza numele bazei de date (se alege un nume pe care o sa il folosim in cadrul urmatorilor pasi) → OK



### PASUL 2:

In Solution Explorer → click dreapta pe sectiunea Connected Services → Add → SQL Server Database

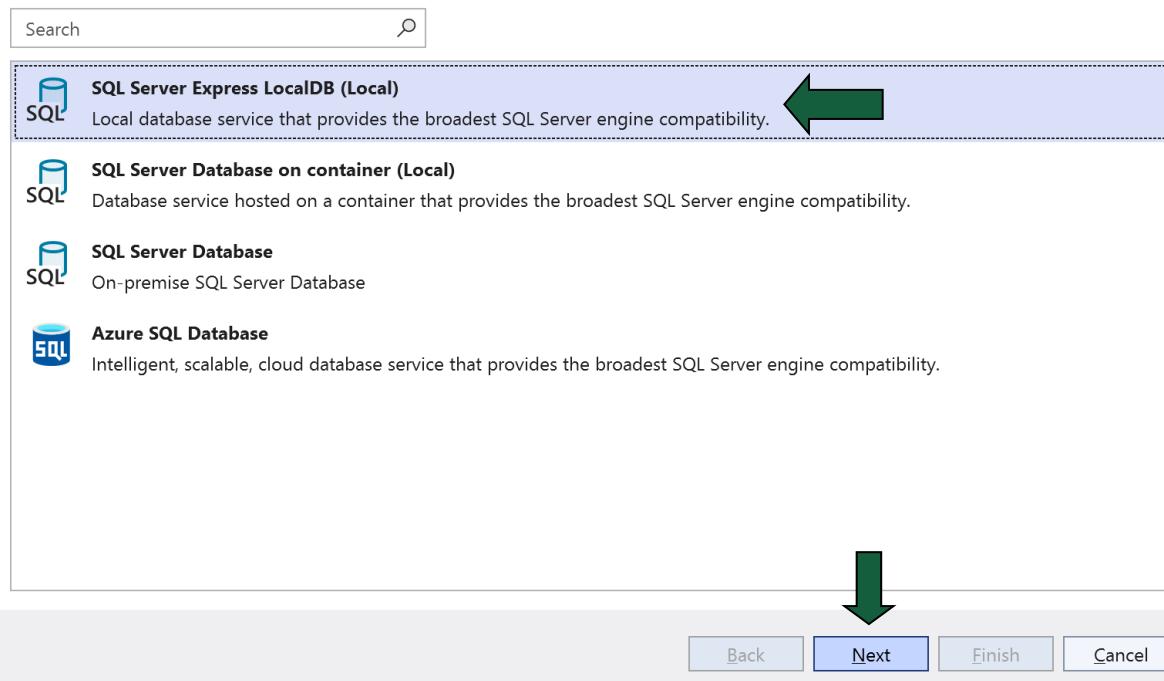


### PASUL 3:

Se selecteaza optiunea SQL Server Express LocalDB (Local)

Connect to dependency

Select a service dependency to add to your application.



### PASUL 4:

Se adauga un nume conexiunii la baza de date (ex: StudentsDB).

Se bifeaza optiunea **None**.

Se modifica string-ul de conexiune apasand cele 3 puncte (unde se afla cercul rosu) din dreptul casutei Connection String Value.

Se apasa butonul Next.

## Connect to SQL Server Express LocalDB (Local)

Provide connection string and specify how to save it

Connection string name

ConnectionString:StudentsDB 

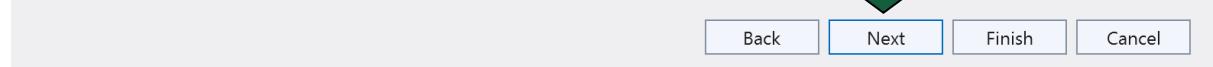
Connection string value

Server=(localdb)\mssqllocaldb;Database=aspnet-53bc9b9d-9d6a-45d4-8429-2a2761773502;Trusted\_Connection=True;MultipleActiveResultSets=True 

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

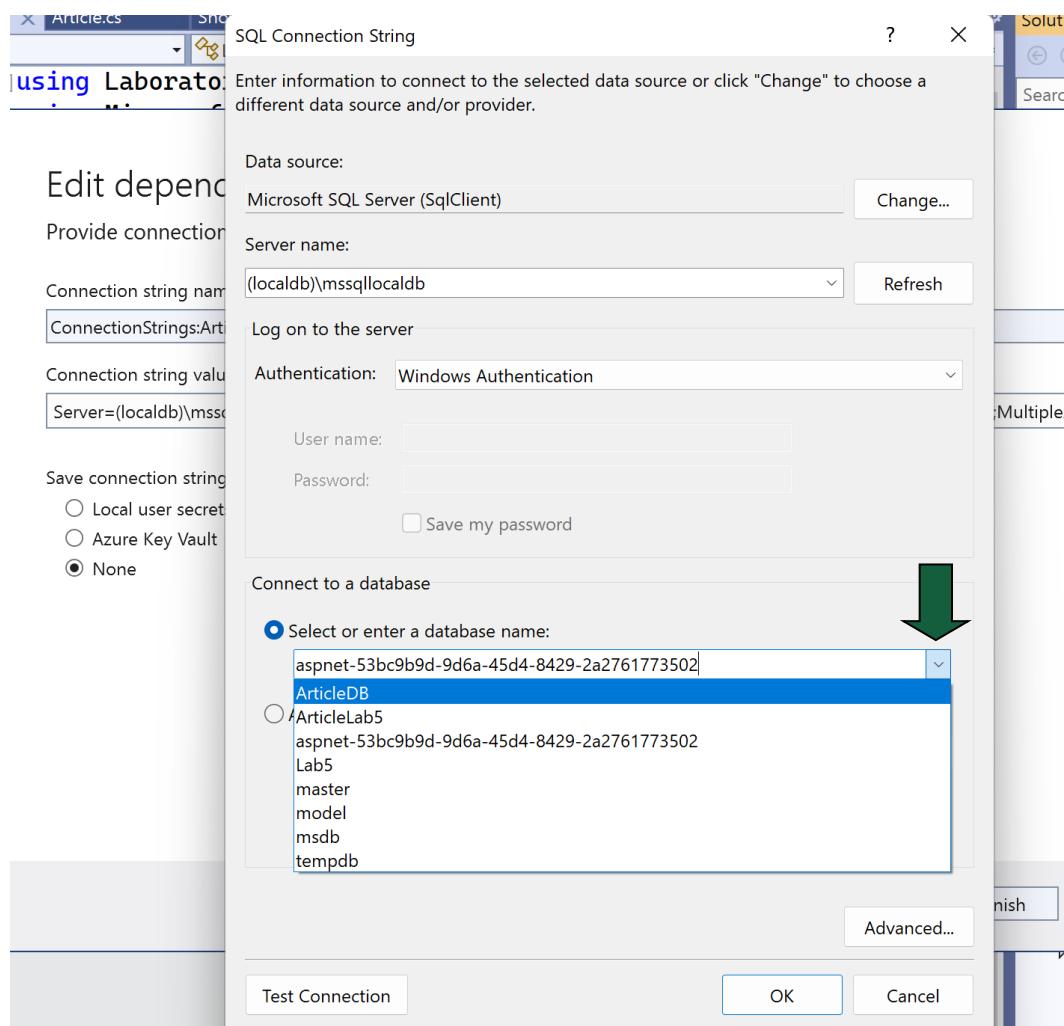
- Local user secrets file
- Azure Key Vault
- None 

 Back Next Finish Cancel



## PASUL 5:

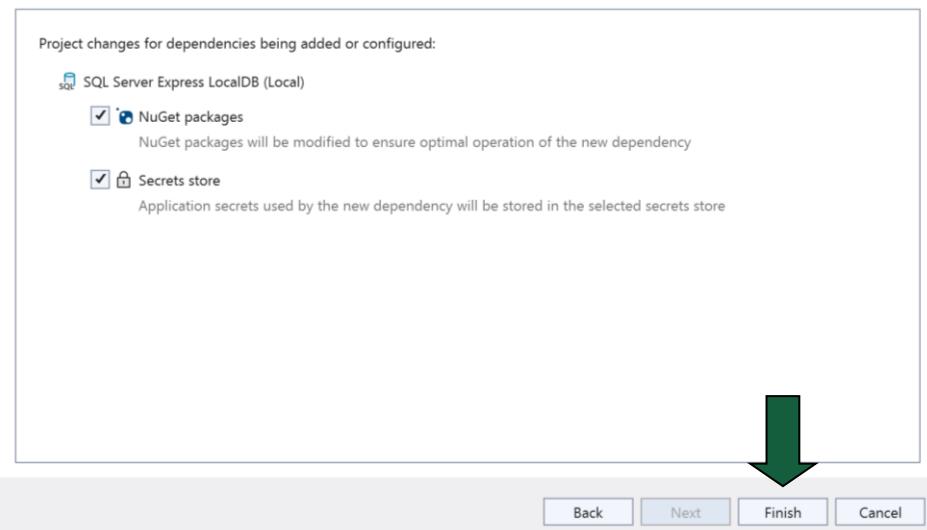
Se selecteaza baza de date creata la Pasul 1.



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

## PASUL 6:

Summary of changes



## PASUL 7:

Se adauga stringul de conectare la baza de date ca parametru al metodei `UseSqlServer()` in `OnConfiguring`

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

Daca se utilizeaza varianta 2 – cea cu Dependency Injection – se adauga stringul de conexiune in **Solution Explorer → appsetting.json** adaugand sevenita de cod marcata cu **BOLD** si acolada.

```
{
  "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": "*"
}

```

### **!\\ OBSERVATIE**

Stringul dat ca parametru in metoda de mai sus UseSqlServer este unic in cadrul fiecarei baze de date. Asadar, nu trebuie sa utilizati stringul din acest exemplu, ci trebuie sa preluati stringul de conexiune la baza voastră de date urmand pasii anterioari.

In acest moment proiectul contine Entity Framework si are creata o baza de date si o conexiune cu aceasta. **Urmeaza sa integram sistemul de migratii.**

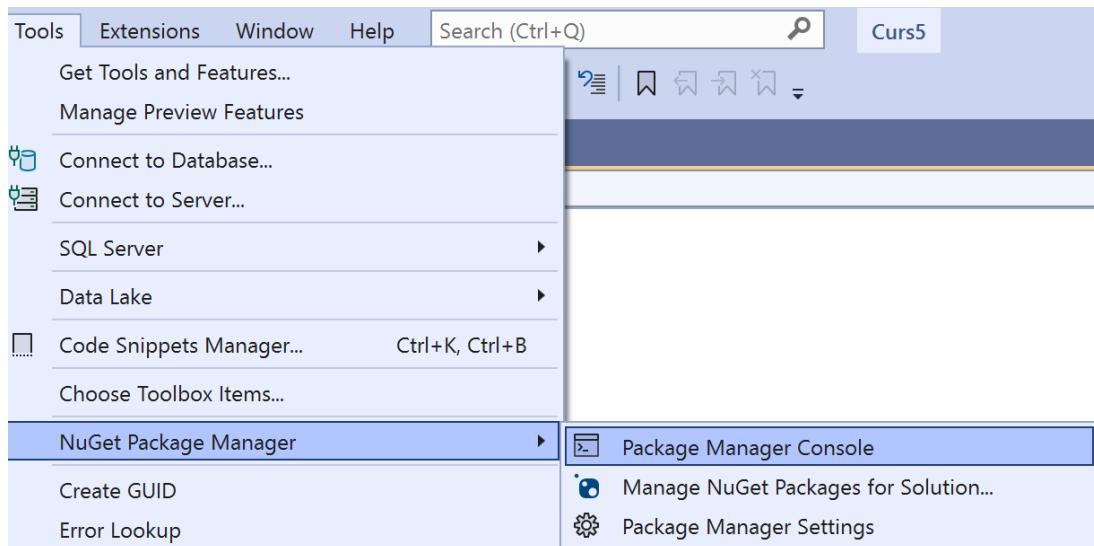
## **Crearea migratiilor in baza de date**

Pentru integrarea sistemului de migratii se parcurg pasii urmatori:

### **PASUL 1:**

Pentru adaugarea migratiilor o sa se utilizeze consola.

Tools → NuGet Package Manager → Package Manager Console



## PASUL 2:

Se adauga migratia utilizand comanda `Add-Migration` urmata de o denumire pe care o dam acestei migratii.

```
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
```

Dupa executarea migratiei, in Solution Explorer o sa se creeze un folder numit **Migrations** unde o sa se genereze fisierele specifice.

## PASUL 3:

Se executa comanda `Update-Database` care modifica baza de date, aducand-o la versiunea finala.



```
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. utilizand Entity Framework

In urmatoarea parte a cursului vom implementa operatiile de tip CRUD asupra entitatii Student, utilizand Entity Framework.

### Index

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

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

    ViewBag.Students = students;
    return View();
}
```

Preluam toti studentii din baza de date, ordonati dupa 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 />
```

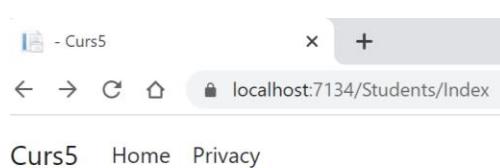
```

        <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>

```



## Afisare studenti

Pop Mihai

pop@exemplu.com

1930101123456

[Afisare student](#)

[Editare student](#)

Popescu Maria

maria@gmail.com

2970202233445

[Afisare student](#)

[Editare student](#)

[Adaugare student](#)

## 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 primara

## 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 adăugare 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

In ASP.NET MVC **model binding** ne permite sa facem legatura intre request-urile de tip HTTP si 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 legatura intre request-urile HTTP si metodele unui Controller (Actiuni). Deoarece datele trimise prin POST sau GET ajung intotdeauna la Controller, acest mecanism de binding leaga in mod automat variabilele de request cu atributele publice ale modelului. Aceasta mapare se va face dupa **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**

### !\\ OBSERVATIE

Este necesar ca numele campurilor din View sa coincida cu numele atributelor pentru ca binding-ul sa functioneze.

## 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>

```

### /!\ OBSERVATIE

In momentul in care sunt necesare in baza de date, fie adaugari sau stergeri de tabele, fie adaugari sau stergeri de coloane sau proprietati, este nevoie de o noua migratie in baza de date.

De exemplu: daca se doreste adaugarea atributului **Address** in clasa Student → `public string Address { get; set; }`

Se adauga proprietatea, dupa care se executa o noua migratie

→ `Add-Migration AddAddressToStudent`

→ `Update-Database`

# Dezvoltarea Aplicatiilor Web utilizand ASP.NET Core MVC

## Laborator 5

### EXERCITII:

#### Exercitiul 1:

Se considera entitatile **Article** si **Category** cu urmatoarele proprietati:

##### **Article**

- ArticleID (int – primary key)
- Title (string)
- Content (string)
- Date (DateTime)

##### **Category:**

- CategoryID (int – primary key)
- CategoryName (string)

Sa se creeze un nou proiect in care sa se adauge doua Controllere **ArticlesController** si **CategoriesController** in care se vor implementa operatiile CRUD asupra entitatilor Article respectiv Category.

**ArticlesController** o sa contina:

- **Index** – pentru afisarea tuturor articolelor din baza de date
- **Show** – pentru afisarea unui singur articol
- **New** – pentru adaugarea unui nou articol in baza de date
- **Edit** – pentru editarea unui articol existent
- **Delete** – pentru stergerea unui articol

**CategoriesController** o sa contina:

- **Index** – pentru afisarea tuturor categoriilor din baza de date
- **Show** – pentru afisarea unei singure categorii
- **New** – pentru adaugarea unei noi categorii in baza de date
- **Edit** – pentru editarea unei categorii existente in baza de date
- **Delete** – pentru stergerea unei categorii

Pasii pe care trebuie sa ii urmati pentru implementarea cerintelor:

1. Creati un nou proiect numit Laborator5 de tip ASP.NET Core Web App (Model-View-Controller).
2. Adaugati Entity Framework Core in proiect (**VEZI Curs 5 – sectiunea Instalare Entity Framework Core**).
3. Adaugati cele doua modele in baza de date → Article si Category (**VEZI Curs 5 – sectiunea Adaugarea Modelului**).

```
public class Article
{
    [Key]
    public int ArticleID { get; set; }

    public string Title { get; set; }

    public string Content { get; set; }

    public DateTime Date { get; set; }
}
```

La fel se procedeaza si pentru clasa Category.

4. Realizati conexiunea cu baza de date (**VEZI Curs 5 – sectiunea Conexiunea cu Baza de Date**).

5. Adaugati baza de date SQL Server (**VEZI Curs 5 – sectiunea Adaugarea unei Baze de Date SQL Server**). Connection String-ul o sa se numeasca **ArticlesDB**.
6. Integrati sistemul de migratii (**VEZI Curs 5 – sectiunea Crearea migratiilor in baza de date**).
7. Implementati CRUD asupra entitatii Article (**VEZI Curs 5 – sectiunea C.R.U.D. utilizand Entity Framework**).
8. Implementati CRUD asupra entitatii Category (**VEZI Curs 5 – sectiunea C.R.U.D. utilizand Entity Framework**).

# Dezvoltarea Aplicatiilor Web utilizand ASP.NET Core MVC

## Curs 6

---

### Cuprins

Baze de Date – Notiuni generale .....	2
Ce este o baza de date .....	2
Ce este un SGBD/DBMS.....	2
Cerinte minimale ale bazelor de date.....	2
Ce este cheia primara .....	3
Ce este cheia externa.....	3
Ce este o cheie primara compusa.....	3
Ce este o entitate .....	3
Ce este o relatie .....	4
Ce este un atribut.....	4
Exemple – Chei, Entitati, Relatii .....	4
Diagrama Entitate/Relatie .....	5
Reguli de proiectare a diagramei E/R .....	5
Exemplu practic pentru proiectarea Diagramei E/R .....	5
Diagrama Conceptuala.....	9
Reguli de transformare a diagramei E/R in Diagrama Conceptuala .....	9
Exemplu pentru proiectarea Diagramei Conceptuale .....	9
Adaugarea sistemului de autentificare .....	11
Implementare cereri folosind Entity Framework Core si LINQ.....	14

# Baze de Date – Notiuni generale

## Ce este o baza de date

O **baza de date** este un ansamblu structurat de date coerente, fara redundanta, astfel incat datele pot fi prelucrate eficient de mai multi utilizatori intr-un mod concurrent.

## Ce este un SGBD/DBMS

Un **SGBD** este un sistem de gestiunea a bazelor de date. Denumirea de SGBD este echivalenta cu cea de DBMS (**Database Management System**) reprezentand un produs software care asigura interactiunea cu o baza de date. Un **DBMS** stocheaza date in tabele, pe care ulterior le acceseaza si prelucreaza cu ajutorul unui limbaj **SQL (Structured Query Language)**. Un DBMS asigura securitatea, integritatea si consistenta datelor.

Exemple de DBMS: Oracle SQL, SQL Server, MySQL.

## Cerinte minime ale bazelor de date

O baza de date trebuie sa indeplineasca urmatoarele cerinte minime:

- Redundanta minima
- Sincronizarea datelor – utilizarea simultana a datelor de catre mai multi utilizatori
- Securitatea datelor
- Integritatea datelor – date corecte, posibilitatea recuperarii lor
- Furnizare rapida a datelor – cereri eficiente
- Flexibilitatea datelor – adaptarea rapida la cerinte noi

## Ce este cheia primara

**Cheia primara** este un identificator **UNIC** in cadrul entitatilor. Ea trebuie sa fie cunoscuta in orice moment – ceea ce inseamna ca **nu poate fi null**

Cu ajutorul cheii primare se identifica unic intrarile dintr-un tabel al bazei de date.

## Ce este cheia externa

**Cheia externa** poate fi ori null in intregime, ori trebuie sa refere cheia primara din tabelul de legatura (adica sa corespunda unei valori a cheii primare asociate).

## Ce este o cheie primara compusa

**Cheia primara compusa** se creeaza in momentul in care se combina doua sau mai multe coloane in cadrul unui tabel, formand un tuplu, care mai apoi este utilizat ca identificator unic in cadrul tabelului respectiv. Tuplul o sa identifice unic fiecare intrare din tabelul in care se afla.

## Ce este o entitate

**Entitate** = un loc, o actiune, o persoana, etc.

**Exemplu de entitati** dintr-o baza de date care gestioneaza o Universitate  
 → Studenti, Cursuri, Note, etc

In modelul Entitate/Relatie, entitatile sunt substantive. In modelele relationale entitatatile devin tabele.

Nu pot exista in aceeasi diagrama doua entitati cu acelasi nume sau aceeasi entitate avand un nume diferit.

## Ce este o relatie

O **relatie** este o asociere dintre doua sau mai multe entitati. In modelul Entitate/Relatie acestea sunt verbe. In modelul relational relatiile devin fie tabele speciale, fie coloane care refera chei primare.

Relatiile au asociata si o cardinalitate, existand trei tipuri de cardinalitati:

- **one-to-many (1:m)**
- **one-to-one (1:1)**
- **many-to-many (m:m)**

## Ce este un atribut

Un **atribut** este o proprietate care descrie o entitate. Fiecare atribut trebuie sa aiba un tip de date, un nume si constrangeri.

## Exemple – Chei, Entitati, Relatii

Un exemplu detaliat se afla in documentul numit **Chei\_Relatii\_Tabele** aflat pe site in sectiunea **Saptamana 6**.

## Diagrama Entitate/Relatie

**Diagrama Entitate/Relatie** este un mod de reprezentare a unui sistem din lumea reală, fiind un model neformalizat. Este compus din entități și relațiile dintre acestea.

### Reguli de proiectare a diagramei E/R

1. Identificarea entităților care fac parte din aplicația pe care dorim să o implementăm
2. Identificarea relațiilor dintre entități și scrierea acestora în cadrul diagramei
3. Identificarea cardinalităților minime și maxime pentru fiecare relație în parte
4. Identificarea atributelor asociate fiecărei entități
5. Stabilirea cheilor primare (atributelor care identifică în mod unic fiecare entitate)

### Exemplu practic pentru proiectarea diagramei E/R

Să se proiecteze o bază de date care modelează o aplicație de tip Engine de stiri având urmatoarele reguli de proiectare:

- să existe cel puțin 4 tipuri de utilizatori: vizitator neînregistrat, utilizator înregistrat, editor și administrator;
- orice utilizator poate vizualiza stările aparute pe site. Pe pagina principală vor apărea stările cele mai recente;
- stările vor fi împărțite pe categorii (create dinamic): știință, tehnologie, sport, etc, existând posibilitatea de adăugare a noi categorii (administratorul poate face CRUD pe categorii);
- stările dintr-o anumita categorie sunt afisate într-o pagină separată unde pot fi sortate după diferite criterii: data apariției și alfabetic;

- editorii se ocupa de publicarea stirilor noi si pot vizualiza, edita, sterge propriile stiri;
- utilizatorii pot adauga comentarii la stirile aparute, isi pot sterge si edita propriile comentarii;
- stirile pot fi cautate prin intermediul unui motor de cautare propriu;
- administratorii se ocupa de buna functionare a intregii aplicatii (ex: pot face CRUD pe stiri, pe categorii, etc.) si pot activa sau revoca drepturile utilizatorilor si editorilor;

## PASUL 1 – Identificarea entitatilor

- sa existe cel putin 4 tipuri de utilizatori → **USER**
- vizitator neinregistrat, utilizator inregistrat, editor si administrator → **ROLES**
- orice utilizator poate vizualiza stirile → **ARTICLES**
- stirile vor fi impartite pe categorii → **CATEGORIES**
- utilizatorii pot adauga comentarii la stirile aparute → **COMMENTS**
- restul functionalitatilor se implementeaza folosind logica de backend (cod), nefiind nevoie de alte entitati/tabele.

### **! OBSERVATIE**

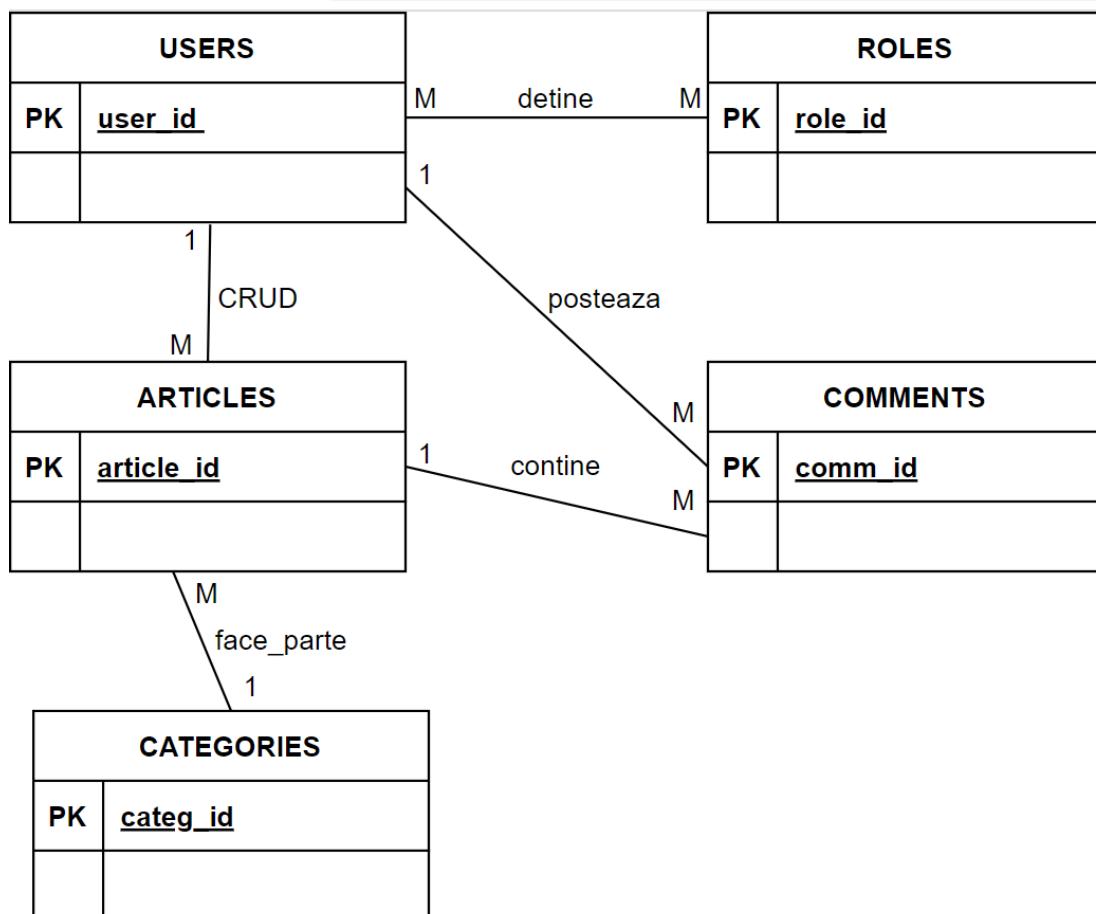
In **Entity Framework** clasele se denumesc la singular deoarece sistemul converteste automat fiecare clasa intr-un tabel, pluralizand numele. Se defineste clasa **Article.cs** in Models, clasa care o sa devina automat tabelul **Articles**.

## PASUL 2 – Identificarea relatiilor si a cardinalitatilor

Avem urmatoarele entitati identificate: **USER, ROLES, ARTICLES, COMMENTS, CATEGORIES**

- sa existe cel putin 4 tipuri de utilizatori: vizitator neinregistrat, utilizator inregistrat, editor si administrator → **un utilizator poate avea un singur rol (! OBS** – avand in vedere ca Entity Framework utilizeaza in acest caz o relatie de tip many-to-many vom implementa si noi o astfel de relatie, dar o vom utiliza ca fiind one-to-many facand asocierea UNUI utilizator cu UN singur rol – ca logica in aplicatie)
- editorii se ocupa de publicarea stirilor noi si pot vizualiza, edita, sterge propriile stiri (un user de tip editor poate sa faca CRUD, iar un user inregistrat poate vizualiza articole) → **un editor vizualizeaza/redacteaza/editeaza/sterge/ mai multe articole, iar un articol este vizualizat/creat/editat/sters/ de un singur editor** → **relatia (cardinalitatea maxima) este one-to-many**  
Pentru vizualizare se foloseste logica scrisa in cod.
- stirile vor fi impartite pe categorii → **un articol face parte dintr-o singura categorie, iar o categorie contine mai multe articole** → **cardinalitatea maxima este one-to-many**
- utilizatorii pot adauga comentarii la stirile aparute, isi pot sterge si edita propriile comentarii → **un articol contine mai multe comentarii, iar un comentariu apartine unui singur articol** → **cardinalitate one-to-many**  
→ **un user posteaza mai multe comentarii, iar un comentariu este postat de un singur user** → **one-to-many**

### PASUL 3- Proiectarea diagramei E/R



## Diagrama Conceptuala

**Diagrama Conceptuala** este un model formal de organizare a datelor, continand legatura dintre acestea sub forma de tabele.

### Reguli de transformare a diagramei E/R in Diagrama Conceptuala

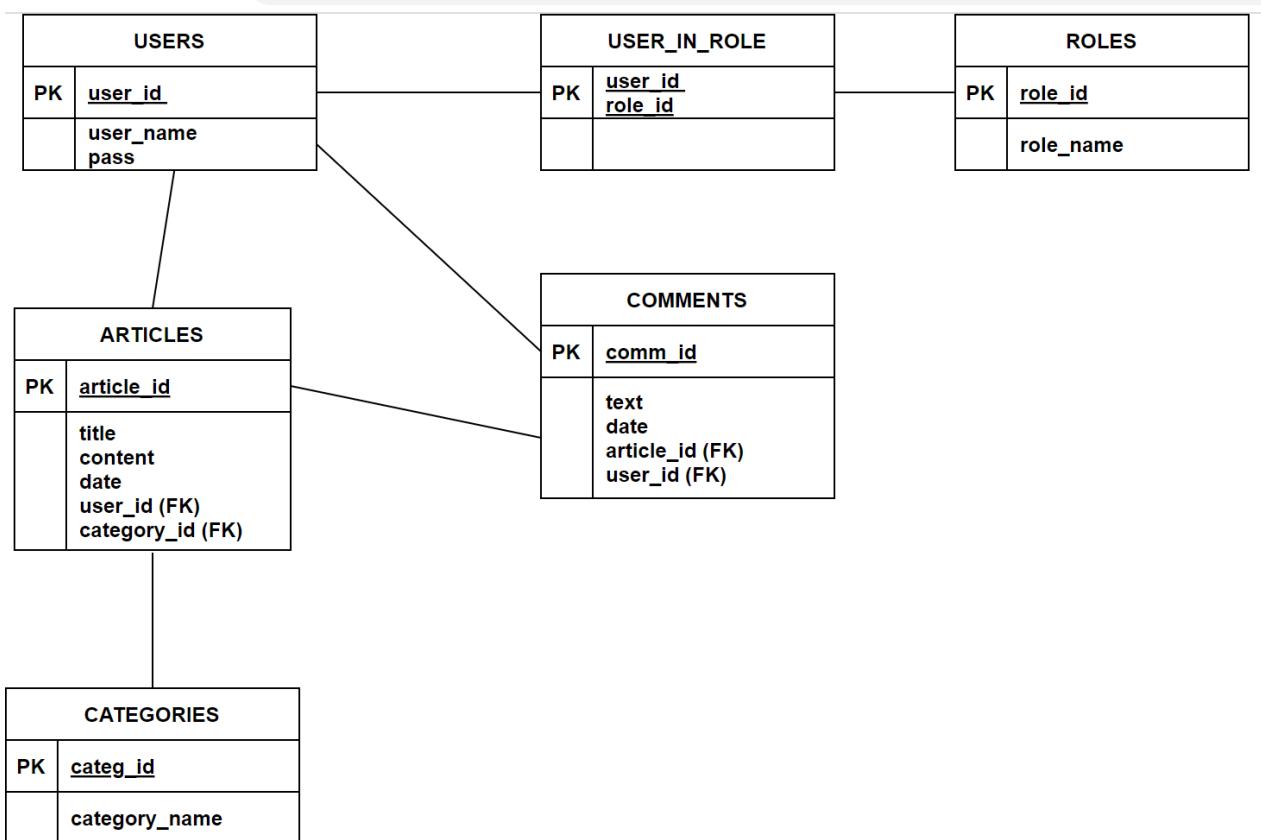
Pentru transformarea Diagramei Entitate/Relatie in Diagrama Conceptuala se urmeaza urmatoarele reguli:

- Entitatile devin tabele;
- Relatiile one-to-one si one-to-many devin chei externe
  - In cazul relatiilor **one-to-many** cheia externa se plaseaza in tabelul in dreptul caruia se afla cardinalitatea **many**
  - In cazul relatiilor **one-to-one** cheia externa se plaseaza in tabelul care contine mai putine intrari in baza de date (din motive de performanta/eficienta)
- Relatiile many-to-many devin tabel asociativ

### Exemplu pentru proiectarea Diagramei Conceptuale

Urmand regulile anterioare transformam Diagrama Entitate/Relatie din sectiunea **Exemplu practic pentru proiectarea Diagramei E/R – PASUL 3** in Diagrama Conceptuala.

## Diagrama Conceptuala:

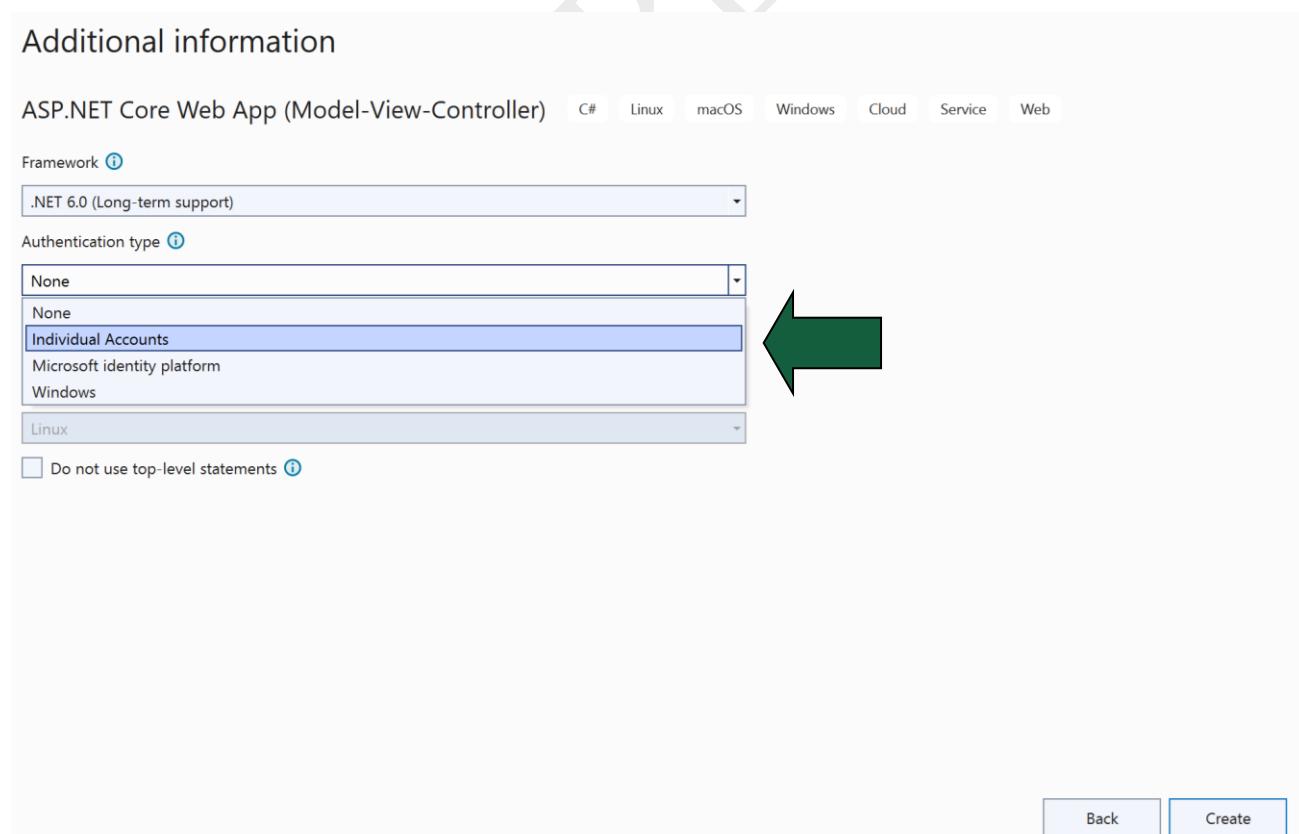


# Adaugarea sistemului de autentificare

Framework-ul ASP.NET Core ofera posibilitatea integrarii unui sistem de autentificare folosind **Identity**.

**Identity** este compus dintr-o suita de clase si sechete de cod care faciliteaza implementarea rapida a unui sistem de autentificare complex. Acest sistem ofera posibilitatea autentificarii folosind user si parola, alocarea de roluri pentru utilizatori, autentificare folosind conturi 3<sub>rd</sub> party (autentificare prin retele de socializare – Google, Facebook, Twitter, etc).

Pentru a genera un proiect care include componenta **Identity** pentru autentificare, trebuie sa alegem la crearea proiectului forma de autentificare: **Individual Accounts**.



Proiectul nou creat contine **Identity Framework** si toate mecanismele aferente autentificarii.

Inainte de rularea proiectului trebuie executata in consola (Tools → NuGet Package Manager → Package Manager Console) comanda **Update-Database**

Dupa rularea comenzii, ne putem inregistra cu un cont.

Laborator6 Home Privacy Register Login

## Register

Create a new account.

Email  
Password  
Confirm password

[Register](#)

Use another service to register.

There are no external authentication services configured. See this [article about setting up this ASP.NET application to support logging in via external services](#).

Dupa inregistrare se apasa [Click here to confirm your account](#) deoarece nu exista inclus serviciul de trimitere de e-mailuri.

Laborator6 Home Privacy Register Login

## Register confirmation

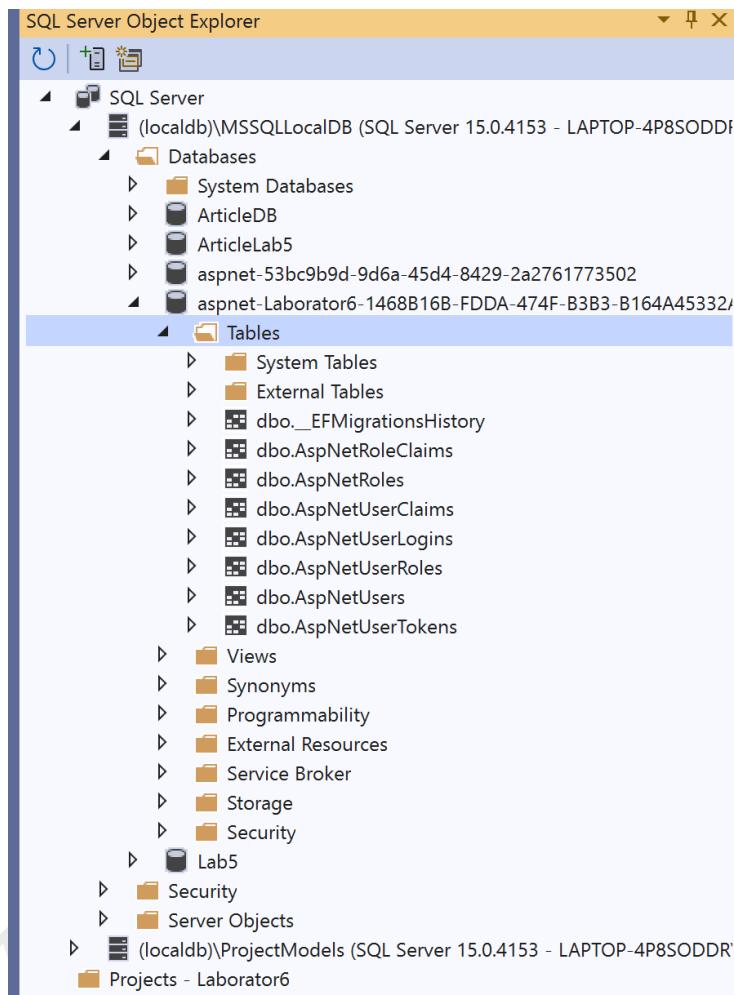
This app does not currently have a real email sender registered, see [these docs](#) for how to configure a real email sender. Normally this would be emailed: [Click here to confirm your account](#)

Laborator6 Home Privacy Register Login

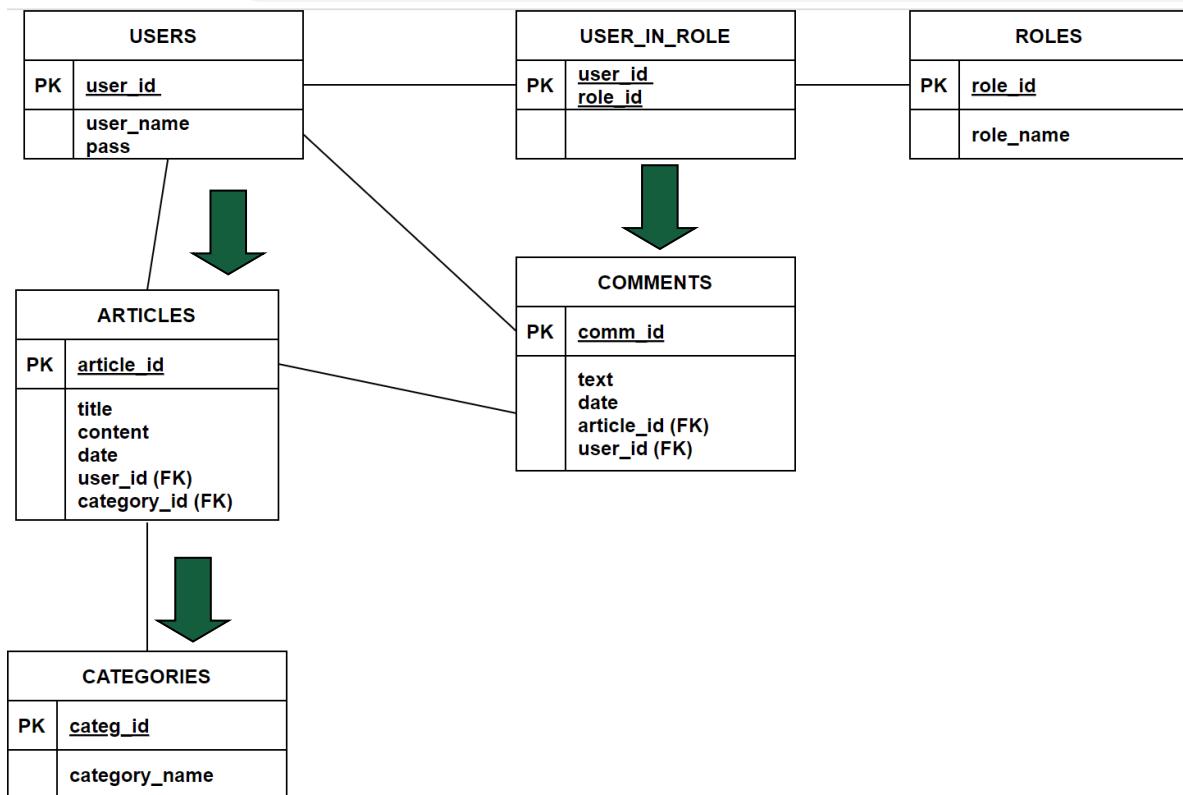
## Confirm email

Thank you for confirming your email. X

In proiect → SQL Server Object Explorer → se pot vizualiza tabelele create pentru sistemul de autentificare si pentru rolurile pe care le pot avea utilizatorii



# Implementare cereri folosind Entity Framework Core si LINQ



Se considera entitatele **Article**, **Category** si **Comment** cu urmatoarele proprietati:

## Article

- Id (int – primary key)
- Title (string – titlul este obligatoriu)
- Content (string – continutul este obligatoriu)
- Date (DateTime)
- CategoryId (int – cheie externa – categoria din care face parte articolul)

### Category:

- Id (int – primary key)
- CategoryName (string – numele este obligatoriu)

### Comment:

- Id (int – primary key)
- Content (string – continutul comentariului este obligatoriu)
- Date (DateTime – data la care a fost postat comentariul)
- Date (DateTime)
- ArticleId (int – cheie externă – articolul caruia ii aparține comentariul)

### Implementarea claselor folosind Entity Framework:

```
public class Article
{
    [Key]
    public int Id { get; set; }

    [Required(ErrorMessage = "Titlul este obligatoriu")]
    public string Title { get; set; }

    [Required(ErrorMessage = "Continutul articolului este
obligatoriu")]
    public string Content { get; set; }

    public DateTime Date { get; set; }

    [Required(ErrorMessage = "Categorija este obligatorie")]
    public int CategoryId { get; set; }

    public virtual Category Category { get; set; }

    public virtual ICollection<Comment> Comments { get; set; }
}
```

```

public class Category
{
    [Key]
    public int Id { get; set; }

    [Required(ErrorMessage = "Numerele categoriei este
obligatoriu")]
    public string CategoryName { get; set; }

    public virtual ICollection<Article> Articles { get; set; }
}

public class Comment
{
    [Key]
    public int CommentId { get; set; }

    [Required(ErrorMessage = "Continutul este obligatoriu")]
    public string Content { get; set; }

    public DateTime Date { get; set; }

    public int ArticleId { get; set; }

    public virtual Article Article { get; set; }
}

```

### ! OBSERVATIE

Dupa implementarea claselor se executa migratiile:

- Add-Migration NumeMigratie
- Update-Database

## JOIN

1. Sa se afiseze articolele impreuna cu categoria din care fac parte

```
var query = db.Articles.Include("Category");
```

unde parametrul **Category** este proprietatea din clasa Article

```
→ public virtual Category Category { get; set; }
```



**Codul asociat generat in consola:**

```
SELECT [a].[Id], [a].[CategoryId], [a].[Content], [a].[Date],
[a].[Title], [c].[Id], [c].[CategoryName]
FROM [Articles] AS [a] INNER JOIN [Categories] AS [c]
ON [a].[CategoryId] = [c].[Id]
```

```
[  
  {  
    "id": 1,  
    "title": "Articol1",  
    "content": "Continut1",  
    "date": "2022-07-11T00:00:00",  
    "categoryId": 1,  
    "category": {  
      "id": 1,  
      "categoryName": "Categ1",  
      "articles": [  
        null  
      ]  
    },  
    "comments": null  
  },  
  {  
    "id": 2,  
    "title": "Articol2",  
    "content": "Continut2",  
    "date": "2022-10-10T00:00:00",  
    "categoryId": 2,  
    "category": {  
      "id": 2,  
      "categoryName": "Categ2",  
      "articles": [  
        null  
      ]  
    },  
    "comments": null  
  }]
```

2. Sa se afiseze articolele, categoria din care fac parte si comentariile asociate. Sa se afiseze titlul articolului, numele categoriei, comentariul si data la care a fost postat comentariul respectiv.

```
var query = from item in
db.Articles.Include("Category").Include("Comments")

select new
{
    ArticleTitle = item.Title,
    CategoryTitle = item.Category.CategoryName,
    Comments = (
        from comm in item.Comments
        select new {
            CommentTitle = comm.Content,
            CommentDate = comm.Date
        }
    )
};
```

→ unde **Category** si **Comments** sunt proprietatile din clasa Article

```
public virtual Category Category { get; set; }
public virtual ICollection<Comment> Comments { get; set; }
```

#### Codul asociat generat in consola:

```
SELECT [a].[Title], [c].[CategoryName], [a].[Id], [c].[Id],
[c0].[Content], [c0].[Date], [c0].[CommentId]
FROM [Articles] AS [a]
INNER JOIN [Categories] AS [c] ON [a].[CategoryId] = [c].[Id]
LEFT JOIN [Comments] AS [c0] ON [a].[Id] = [c0].[ArticleId]
ORDER BY [a].[Id], [c].[Id]
```

```

[ {
    "articleTitle": "Articol1",
    "categoryTitle": "Categ1",
    "comments": [
        {
            "commentTitle": "Comm1",
            "commentDate": "2022-07-11T00:00:00"
        },
        {
            "commentTitle": "Comm2",
            "commentDate": "2022-07-02T00:00:00"
        }
    ],
    {
        "articleTitle": "Articol2",
        "categoryTitle": "Categ2",
        "comments": []
    }
}
]
  
```

Acste proprietati sunt **virtuale** pentru a se executa **lazy loading**. Lazy loading este un design pattern folosit in dezvoltarea web care intarzie incarcarea unumitor resurse pentru a imbunatati performanta.

In cazul nostru, lazy loading se realizeaza prin declararea proprietatilor de tip virtual si duce la incarcarea claselor doar in momentul in care sunt folosite proprietatile respective. Un exemplu in acest sens ar fi utilizarea joinului folosind **Include**, dar nefolosirea proprietatii Category. In acest caz vom observa ca indiferent daca exista in join, nefiind utilizata pentru afisare in query, nu o sa fie incarcata proprietatea Category din cadrul clasei Article.

```

var query = from item in
db.Articles.Include("Category").Include("Comments")

    select new
    {
        ArticleTitle = item.Title,
        //CategoryTitle = item.Category.CategoryName,
        Comments = (
            from comm in item.Comments
            select new {
                CommentTitle = comm.Content,
                CommentDate = comm.Date
            }
        )
    };

```

### Codul asociat generat in consola:

```

SELECT [a].[Title], [a].[Id], [c].[Content], [c].[Date],
[c].[CommentId]
FROM [Articles] AS [a]
LEFT JOIN [Comments] AS [c] ON [a].[Id] = [c].[ArticleId]
ORDER BY [a].[Id]

```



The screenshot shows a JSON structure representing a list of articles and their associated comments. The root is an array of objects, each representing an article. Each article object contains the 'articleTitle' and 'comments' arrays.

```

[
  {
    "articleTitle": "Articol1",
    "comments": [
      {
        "commentTitle": "Comm1",
        "commentDate": "2022-07-11T00:00:00"
      },
      {
        "commentTitle": "Comm2",
        "commentDate": "2022-07-02T00:00:00"
      }
    ]
  },
  {
    "articleTitle": "Articol2",
    "comments": []
  }
]

```

## GROUP BY si functii grup (de exemplu COUNT)

Pentru fiecare categorie sa se afiseze numarul de articole fac parte din categoria respectiva. O sa se afiseze id-ul si numele categoriei, impreuna cu numarul de articole care fac parte din categoria respectiva.

```
var query = from category in db.Categories
            join article in db.Articles
            on category.Id equals article.CategoryId
            group category by category.Id into groupedCategories

            select new
            {
                CategoryId = groupedCategories.Key,
                ArticlesCount = groupedCategories.Count(),
                CategorySpecificSelection = from _item in
groupedCategories
                    select new
                    {
                        CategoryName = _item.CategoryName,
                    }
            };
        }
```

### Codul generat in consola:

```
SELECT [t].[Id], [t].[c], [t0].[CategoryName], [t0].[Id],
[t0].[Id0]
FROM (
    SELECT [c].[Id], COUNT(*) AS [c]
    FROM [Categories] AS [c]
    INNER JOIN [Articles] AS [a] ON [c].[Id] =
[a].[CategoryId]
    GROUP BY [c].[Id]
) AS [t]
LEFT JOIN (
    SELECT [c0].[CategoryName], [c0].[Id], [a0].[Id] AS
[Id0]
    FROM [Categories] AS [c0]
    INNER JOIN [Articles] AS [a0] ON [c0].[Id] =
[a0].[CategoryId]
) AS [t0] ON [t].[Id] = [t0].[Id]
ORDER BY [t].[Id], [t0].[Id]
```

```
[  
  {  
    "categoryId": 1,  
    "articlesCount": 1,  
    "categorySpecificSelection": [  
      {  
        "categoryName": "Categ1"  
      }  
    ],  
  },  
  {  
    "categoryId": 2,  
    "articlesCount": 2,  
    "categorySpecificSelection": [  
      {  
        "categoryName": "Categ2"  
      },  
      {  
        "categoryName": "Categ2"  
      }  
    ]  
  }  
]
```

In acelasi mod se pot selecta in continuare informatiile despre articolele asociate, adaugand urmatoarea linie de cod in secventa anterioara:

```
var query = from category in db.Categories  
           join article in db.Articles  
           on category.Id equals article.CategoryId  
           group category by category.Id into groupedCategories  
  
           select new  
           {  
             CategoryId = groupedCategories.Key,  
             ArticlesCount = groupedCategories.Count(),  
             CategorySpecificSelection = from _item in  
groupedCategories  
               select new  
               {  
                 CategoryName = _item.CategoryName,  
                 CategoryArticles = _item.Articles  
               }  
           };
```



### Codul generat în consola:

```

SELECT [t].[Id], [t].[c], [t0].[CategoryName], [t0].[Id],
[t0].[Id0], [t0].[Id1], [t0].[CategoryId], [t0].[Content],
[t0].[Date], [t0].[Title]
FROM (
    SELECT [c].[Id], COUNT(*) AS [c]
    FROM [Categories] AS [c]
    INNER JOIN [Articles] AS [a] ON [c].[Id] =
[a].[CategoryId]
    GROUP BY [c].[Id]
) AS [t]
LEFT JOIN (
    SELECT [c0].[CategoryName], [c0].[Id], [a0].[Id] AS
[Id0], [a1].[Id] AS [Id1], [a1].[CategoryId], [a1].[Content],
[a1].[Date], [a1].[Title]
    FROM [Categories] AS [c0]
    INNER JOIN [Articles] AS [a0] ON [c0].[Id] =
[a0].[CategoryId]
    LEFT JOIN [Articles] AS [a1] ON [c0].[Id] =
[a1].[CategoryId]
    ) AS [t0] ON [t].[Id] = [t0].[Id]
ORDER BY [t].[Id], [t0].[Id], [t0].[Id0]

```

```

[
  {
    "categoryId": 1,
    "articlesCount": 1,
    "categorySpecificSelection": [
      {
        "categoryName": "Categ1",
        "categoryArticles": [
          {
            "id": 1,
            "title": "Articol1",
            "content": "Continut1",
            "date": "2022-07-11T00:00:00",
            "categoryId": 1,
            "category": null,
            "comments": null
          }
        ]
      }
    ]
  }
]

```

```

    {
      "categoryId": 2,
      "articlesCount": 2,
      "categorySpecificSelection": [
        {
          "categoryName": "Categ2",
          "categoryArticles": [
            {
              "id": 2,
              "title": "Articol2",
              "content": "Continut2",
              "date": "2022-10-10T00:00:00",
              "categoryId": 2,
              "category": null,
              "comments": null
            },
            {
              "id": 5,
              "title": "Articol3",
              "content": "Continut3",
              "date": "2022-10-11T00:00:00",
              "categoryId": 2,
              "category": null,
              "comments": null
            }
          ]
        }
      ]
    }
  
```

In cazul in care se doreste implementarea unei **relatii de tip many-to-many**, framework-ul genereaza automat tabelul asociativ.

**De exemplu:** daca avem clasele Student si Course, impreuna cu relatiile:

- un student participa la mai multe cursuri
- in cadrul unui curs participa multi studenti

Este suficient ca:

- in clasa **Student** sa existe o proprietate
  - **public virtual ICollection <Course> Courses { get; set; }**
- iar in clasa **Course** sa existe o proprietate:
  - **public virtual ICollection <Student> Students { get; set; }**

# Dezvoltarea Aplicatiilor Web utilizand ASP.NET Core MVC

## Laborator 6

---

### EXERCITII:

1. Sa se creeze un proiect, in cadrul caruia sa se adauge Entity Framework, baza de date si sa se includa sistemul de migratii (**VEZI Curs 5 – Sectiunea Crearea unui proiect utilizand EF si sistemul de migratii**).
  
2. Sa se testeze corectitudinea pasilor implementati in cadrul exercitiului 1 adaugand clasa Articles.cs, iar in Controller-ul ArticlesController sa se creeze metoda Index, metoda prin care se afiseaza toate articolele din baza de date. De asemenea, o sa fie nevoie si de un folder Articles in care o sa se creeze View-ul Index.cshtml.

- a. Se adauga Clasa → Article.cs

```
public class Article
{
    [Key]
    public int ArticleID { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public DateTime Date { get; set; }
}
```

- b. Se ruleaza sistemul de migratii pentru update-ul bazei de date
  
- c. Se deschide tabelul si se insereaza 2 intrari in tabel
  
- d. Se implementeaza metoda Index in cadrul Controller-ului si View-ul asociat in folderul din View.

## Metoda Index din ArticlesController

```
public IActionResult Index()
{
    var articles = from article in db.Articles
                   select article;

    ViewBag.Articles = articles;

    return View();
}
```

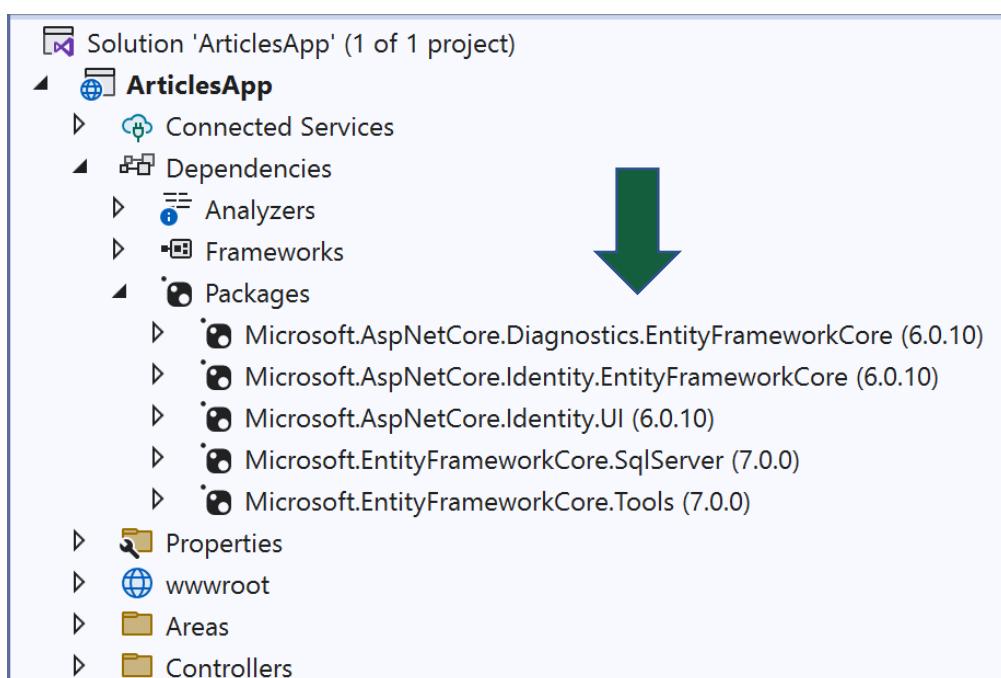
## View-ul Index.cshtml

```
<h2>Afisare articole</h2>
<br />

@foreach (var art in ViewBag.Articles)
{
    <p>@art.Title</p>
    <p>@art.Content</p>
    <p>@art.Date</p>

    <br />
    <a href="/Articles>Show/@art.ArticleID">Afisare
    articol</a>
    <br />
    <a href="/Articles>Edit/@art.ArticleID">Editare
    articol</a>
    <hr />
}
```

3. Sa se creeze un nou proiect numit **ArticlesApp** de tipul ASP.NET Core Web App (Model-View-Controller). Proiectul o sa contine sistem de autentificare (**VEZI Curs 6 – Secțiunea Adaugarea Sistemului de Autentificare**).
  
4. Se verifica existenta urmatoarelor pachete:



5. Se ruleaza migratia Update-Database pentru realizarea update-ului in baza de date. Dupa acest pas se pot vedea tabelele in SQL Server Object Explorer. Se ruleaza doar comanda Update-Database deoarece migratia initiala exista.
  
6. Se ruleaza proiectul si se inregistreaza un cont, dupa care se poate vizualiza noul user in baza de date, tabelul **dbo.AspNetUsers**.

7. Se considera entitatile **Article**, **Category** si **Comment** cu proprietatile de mai jos. De asemenea, se considera cerintele din cadrul **Cursului 6 – sectiunea Exemplu practic pentru proiectarea Diagramei E/R**.

### **Article**

- Id (int – primary key)
- Title (string – titlul este obligatoriu)
- Content (string – continutul este obligatoriu)
- Date (DateTime)
- CategoryId (int – cheie externa – categoria din care face parte articolul)

### **Category:**

- Id (int – primary key)
- CategoryName (string – numele este obligatoriu)

### **Comment:**

- Id (int – primary key)
- Content (string – continutul comentariului este obligatoriu)
- Date (DateTime – data la care a fost postat comentariul)
- Date (DateTime)
- ArticleId (int – cheie externa – articolul caruia ii apartine comentariul)

- sa existe cel putin 4 tipuri de utilizatori: vizitator neinregistrat, utilizator inregistrat, editor si administrator;
- orice utilizator poate vizualiza stirile aparute pe site. Pe pagina principala vor aparea stirile cele mai recente;
- stirile vor fi impartite pe categorii (create dinamic): stiinta, tehnologie, sport, etc, existand posibilitatea de adaugare a noi categorii (administratorul poate face CRUD pe categorii);
- stirile dintr-o anumita categorie sunt afisate intr-o pagina separata unde pot fi sortate dupa diferite criterii: data aparitiei si alfabetic;
- editorii se ocupa de publicarea stirilor noi si pot vizualiza, edita, sterge propriile stiri;

- utilizatorii pot adauga comentarii la stirile aparute, isi pot sterge si edita propriile comentarii;
- stirile pot fi cautate prin intermediul unui motor de cautare propriu;
- administratorii se ocupa de buna functionare a intregii aplicatii (ex: pot face CRUD pe stiri, pe categorii, etc.) si pot activa sau revoca drepturile utilizatorilor si editorilor;

Sa se implementeze cele trei clase in Models, dupa care sa se ruleze migratiile → in acest caz Add-Migration NumeMigratie si Update-Database (**VEZI Pasul urmator – exercitiul 9**).

### Implementarea claselor:

```
public class Article
{
    [Key]
    public int Id { get; set; }

    [Required(ErrorMessage = "Titlul este obligatoriu")]
    public string Title { get; set; }

    [Required(ErrorMessage = "Continutul articolului este
obligatoriu")]
    public string Content { get; set; }

    public DateTime Date { get; set; }

    [Required(ErrorMessage = "Categoria este obligatorie")]
    public int CategoryId { get; set; }

    public virtual Category Category { get; set; }

    public virtual ICollection<Comment> Comments { get; set; }
}
```

```

public class Category
{
    [Key]
    public int Id { get; set; }

    [Required(ErrorMessage = "Numele categoriei este obligatoriu")]
    public string CategoryName { get; set; }

    public virtual ICollection<Article> Articles { get; set; }
}

public class Comment
{
    [Key]
    public int CommentId { get; set; }

    [Required(ErrorMessage = "Continutul este obligatoriu")]
    public string Content { get; set; }

    public DateTime Date { get; set; }

    public int ArticleId { get; set; }

    public virtual Article Article { get; set; }
}

```

8. Se adauga proprietatile in contextul bazei de date pentru realizarea ulterioara a migratiilor.

```

public DbSet<Article> Articles { get; set; }
public DbSet<Category> Categories { get; set; }
public DbSet<Comment> Comments { get; set; }

```

9. Sa se ruleze sistemul de migratii, dupa care sa se insereze manual in fiecare tabel 2-3 intrari (in acest caz se ruleaza ambele comenzi).

10. Sa se adauge cate un Controller pentru fiecare clasa → **ArticlesController**, **CategoriesController** si **CommentsController** in care se vor implementa operatiile CRUD asupra entitatilor.

11.Sa se adauge cate un folder in Views pentru fiecare Controller.

12.Implementati operatii CRUD asupra entitatilor, urmand pasii urmatori.

➤ Sa existe posibilitatea realizarii operatiilor **C.R.U.D. pentru entitatea Article** astfel:

- **Index** – afisarea tuturor articolelor, impreuna cu denumirea categoriei din care fac parte – se poate utiliza din Bootstrap -> Cards: <https://getbootstrap.com/docs/5.2/components/card/#about>
- **Show** – afisarea unui singur articol (intr-o pagina separata) impreuna cu denumirea categoriei din care face parte articolul respectiv
- **New** – posibilitatea adaugarii unui nou articol. In momentul in care se adauga articolul, categoria se va selecta dintr-o lista existenta de categorii, folosind un element de tipul dropdown
- **Edit** – posibilitatea editarii unui articol. In momentul in care se editeaza articolul, categoria se va selecta dintr-o lista existenta de categorii (element de tipul dropdown)
- **Delete** – posibilitatea stergerii unui articol

➤ Sa existe posibilitatea realizarii operatiilor **C.R.U.D. pentru entitatea Category** (**Atentie!** In momentul in care se doreste stergerea unei categorii, automat se vor sterge si toate articolele care fac parte din categoria respectiva).

➤ Sa existe posibilitatea realizarii operatiilor **C.R.U.D. pentru entitatea Comment** astfel:

- **Afisarea tuturor comentariilor** corespunzatoare unui articol. Fiecare articol o sa aiba un buton “Afisare articol” (Show) care redirectioneaza catre o pagina in care vom avea articolul impreuna cu toate comentariile corespunzatoare articolului respectiv

Pentru afisarea articolelor se poate utiliza din Bootstrap > Cards:

<https://getbootstrap.com/docs/5.2/components/card/#about>

- **New** – posibilitatea adaugarii unui nou comentariu
- **Edit** – posibilitatea editarii unui comentariu existent
- **Delete** – posibilitatea stergerii unui comentariu

### **Surse utile:**

**Bootstrap – v5.2** → <https://getbootstrap.com/docs/5.2/getting-started/introduction/>

**Bootstrap – icons** → <https://icons.getbootstrap.com/>

**Bootstrap – buttons** →  
<https://getbootstrap.com/docs/5.2/components/buttons/>

**Bootstrap – spacing** →  
<https://getbootstrap.com/docs/5.0/utilities/spacing/>

**Flexbox** → <https://css-tricks.com/snippets/css/a-guide-to-flexbox/>

# Dezvoltarea Aplicatiilor Web utilizand ASP.NET Core MVC

## Curs 7

---

### Cuprins

View (interfata cu utilizatorul).....	2
Ce este View-ul.....	2
Razor in cadrul unui proiect ASP.NET Core MVC .....	4
Crearea unui View in cadrul unui proiect .....	7
Trimiterea datelor catre View .....	9
Model .....	9
ViewBag.....	10
ViewData .....	11
TempData.....	12
Helpere pentru View .....	17
Implementarea alternativa a Helperelor.....	23

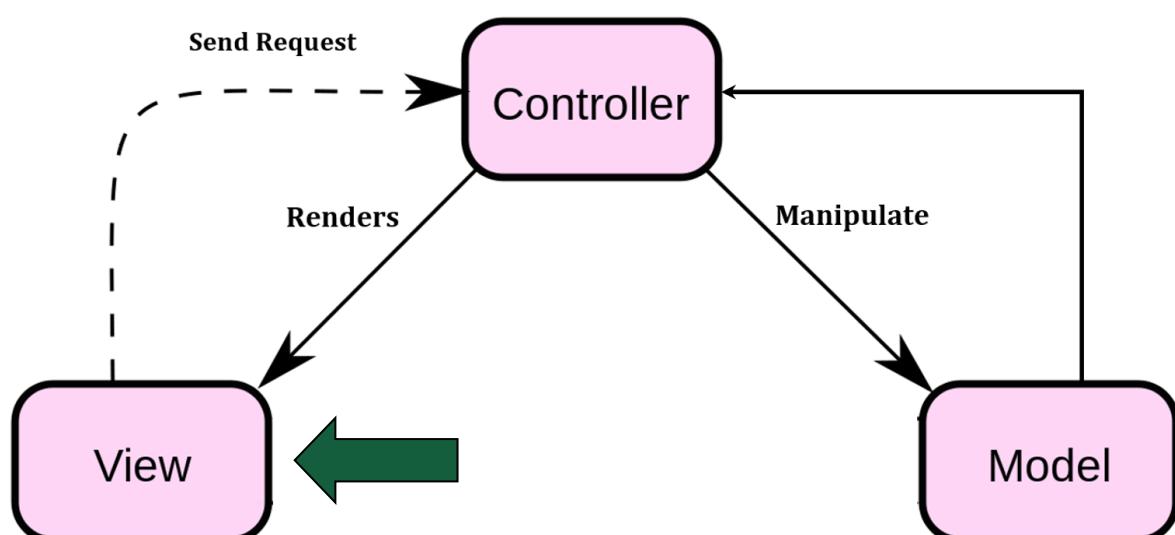
# View (interfata cu utilizatorul)

## Ce este View-ul

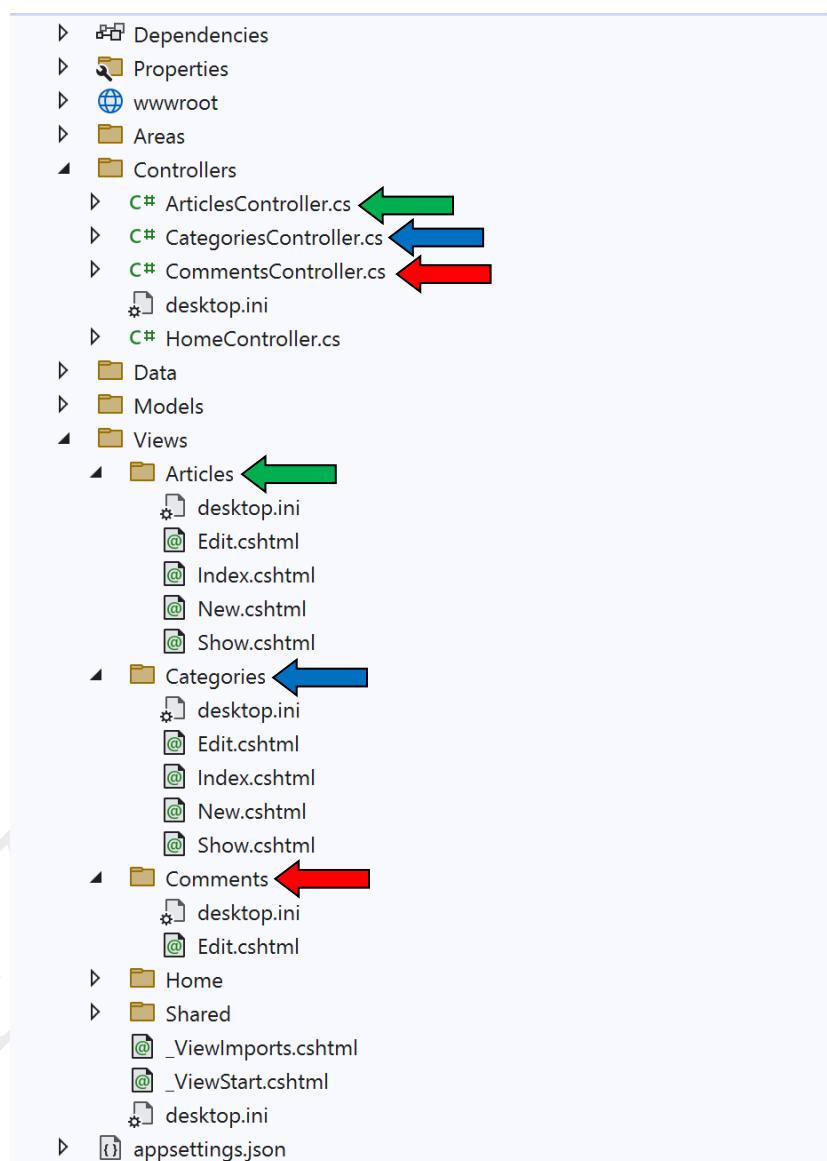
**View-ul** reprezinta interfata cu utilizatorul, fiind componenta arhitecturii MVC cu care utilizatorii interactioneaza prin intermediul browser-ului.

In View se afiseaza datele, adica inregistrarile din baza de date si informatiile generate de aplicatie.

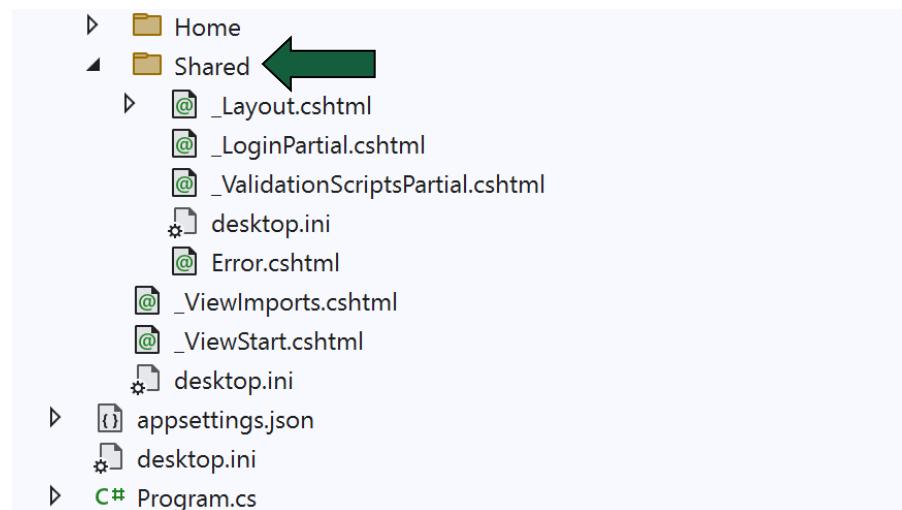
View-ul poate contine HTML static sau chiar HTML trimis din Controller (HTML dinamic). In cadrul arhitecturii MVC, View-ul comunica doar cu Controller-ul, iar cu Modelul comunica indirect tot prin intermediul Controller-ului.



View-urile in ASP.NET Core MVC se afla in folderul **Views**. Diferitele actiuni (metode) implementate intr-un Controller randeaza diferite View-uri, ceea ce inseamna ca folderul Views contine cate un folder separat pentru fiecare Controller, cu acelasi nume ca si Controller-ul, dupa cum se observa in imaginea urmatoare:



Folderul **Shared** contine View-urile (layout-uri, view-uri partiale) care sunt partajate (folosite) in cadrul mai multor View-uri existente in aplicatie.



## Razor in cadrul unui proiect ASP.NET Core MVC

Inca din ASP.NET MVC 3 motorul (engine-ul) de afisare a View-urilor in acest framework este **Razor**. Acest motor ne ofera posibilitatea de a mixa tag-uri HTML cu cod C#. In Razor se utilizeaza caracterul @ pentru a incepe o secventa de cod **server-side**.

**Sintaxa Razor** este simpla si a fost construita cu scopul de a minimiza lungimea codului scris. Aceasta este foarte compacta, usor de invatat si este suportata de editorul Visual Studio.

### Exemple sintaxa Razor:

- Se foloseste **@** pentru executarea codului server-side pe o singura linie (de exemplu: pentru afisarea valorii unei variabile).

**Exemplu** – se afiseaza valoarea variabilei **ViewBag.Article** trimisa din Controller in View, preluandu-se atributul Title din modelul Article (conform exemplului implementat in cadrul laboratorului)

```
<h3>@ViewBag.Article.Title</h3>
```

- Parcursarea colectiei **ViewBag.Articles** pentru afisarea articolelor

```
@foreach (var article in ViewBag.Articles)
```

- Pentru a executa un bloc intreg de cod (mai multe linii de cod) sau pentru a da o valoare unei variabile este necesar sa folosim accolade, astfel: `@{ /* cod */ }`

**Exemplu** – variabila ViewBag.Title primeste valoarea “Index”

```
@{
    ViewBag.Title = "Index";
}
```

Cu alte cuvinte, atunci cand se integreaza cod C# in HTML, se utilizeaza simbolul `@` pentru o singura linie de cod si `@{ }`  pentru o secventa de cod.

- In cadrul unei secvente de cod se poate include cod HTML si text utilizand simbolul `@`:

**Exemplul 1** – pentru includerea textului:

```
@if(1 > 0)
{
    @:este adevarat
}
```

In acest exemplu, putem vedea cum se poate afisa o secventa de text in cadrul unui bloc de cod. Secventa de mai sus va afisa pe ecran mesajul “este adevarat”.

**Exemplul 2** – pentru includerea codului HTML (exemplu preluat din Laborator 6):

```
<select name="CategoryId">
    @foreach (var item in ViewBag.Categories)
    {
        <option value="@item.Id">@item.CategoryName</option>
    }
</select>
```

➤ Conditia **if** incepe cu: **@if{ ... }**

**Exemplu:**

```
@if(1 > 10)
{
    @:este fals
}
```

➤ Loop-ul are urmatoarea sintaxa: **@for{ ... }**

**Exemplu:**

```
@for (int i = 0; i < 10; i++)
{
    @i.ToString()
```

- **@model** ofera posibilitatea afisarii valorilor modelului oriunde in View

### **Exemplu:**

```
@model Curs7.Models.Student – includerea se realizeaza in
View-ul asociat metodei
. . .
<h1>@Model.Name</h1>
<p>@Model.Email</p>
<p>@Model.CNP</p>
```

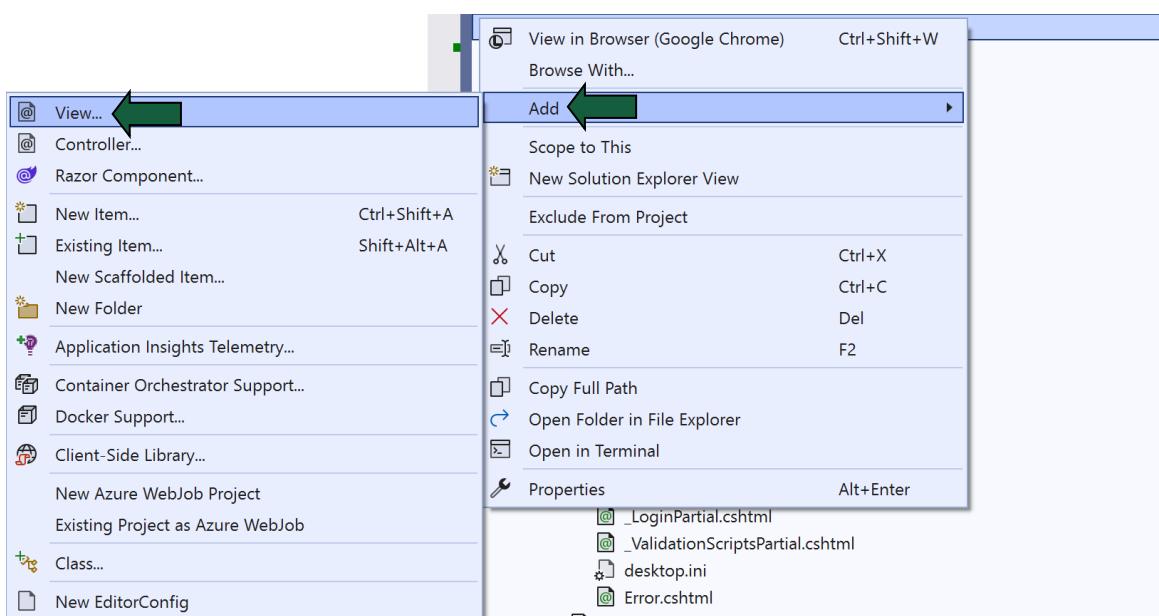
## **Crearea unui View in cadrul unui proiect**

Se creeaza un View dupa cum urmeaza:

### **PASUL 1:**

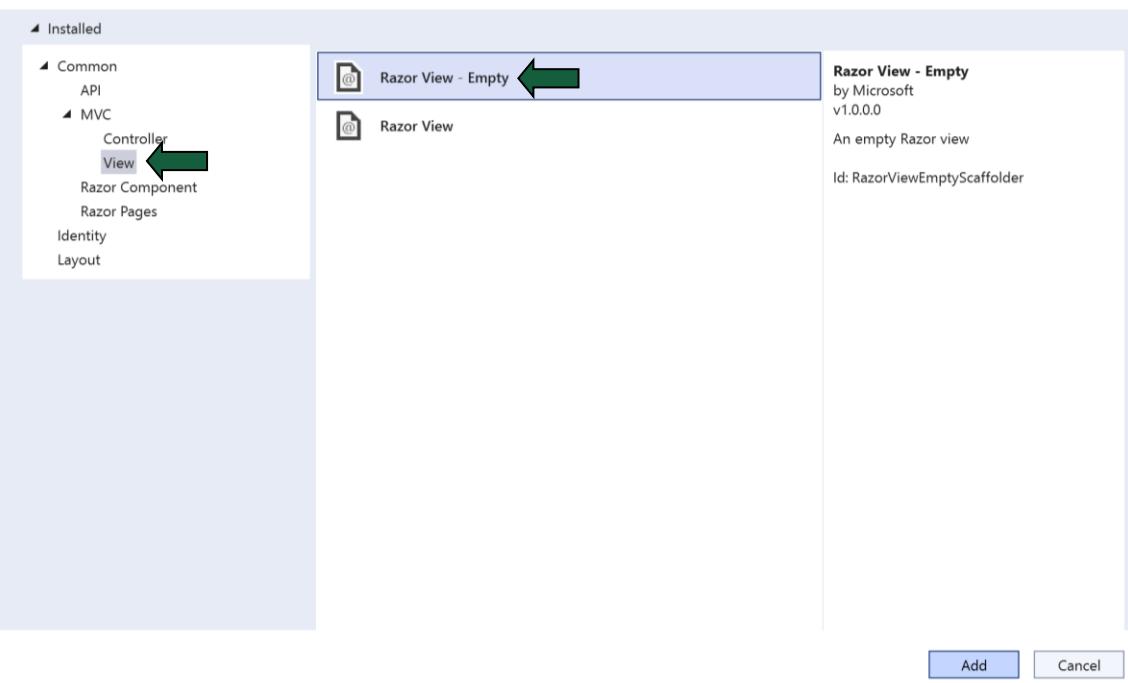
**Click dreapta** pe folderul corespunzator din folderul View → **Add** → **View**.

**De exemplu:** pentru prelucrarea articolelor din baza de date (**VEZI Laborator 6**) o sa avem Controller-ul ArticlesController, iar in folderul Views trebuie creat folderul Articles asociat. In acesta o sa existe View-urile asociate cu merocele din Controller.

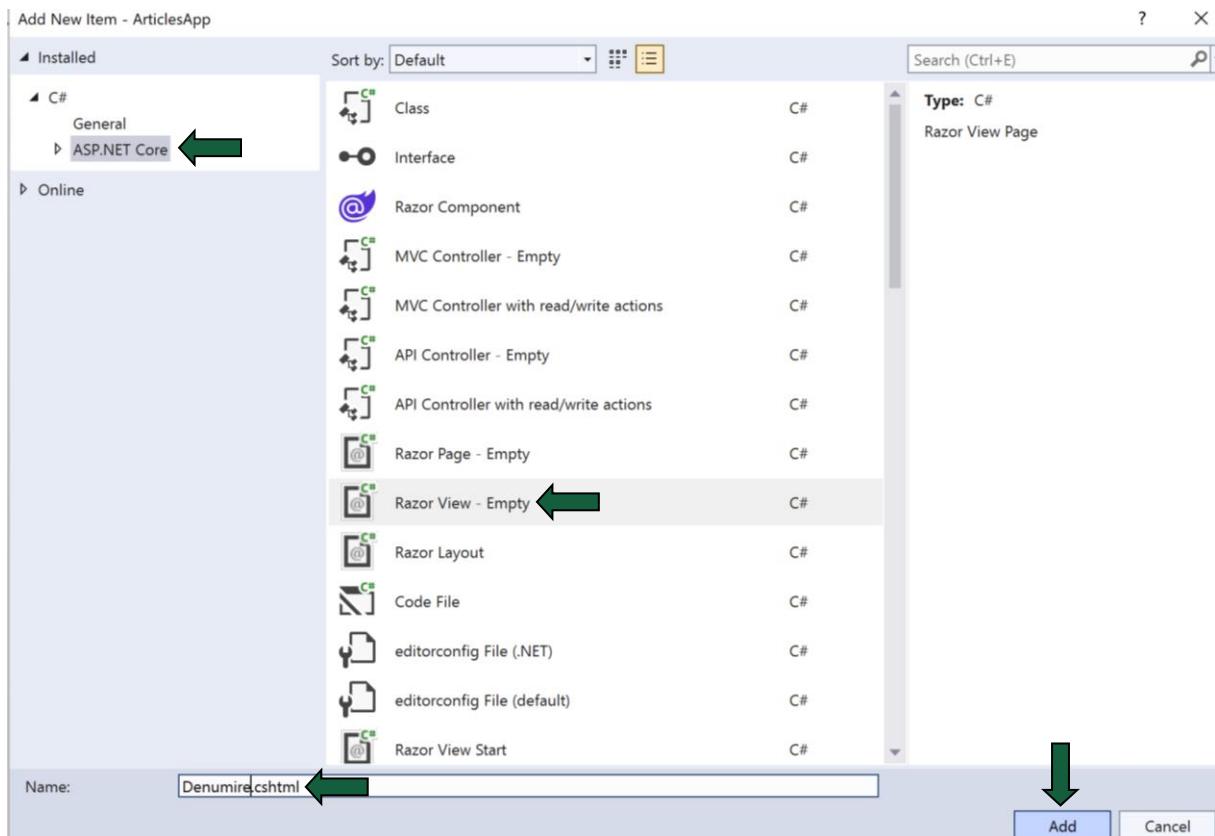


## PASUL 2:

### Add New Scaffolded Item



## PASUL 3:



## Trimiterea datelor catre View

De foarte multe ori este necesar sa trimitem diferite date catre View pentru afisarea acestora in browser. Pe langa informatiile primite din baza de date (prin intermediul modelului) exista cazuri in care vom trimite si alte tipuri de date.

## Model

**Trimiterea datelor din baza de date** se face prin intermediul helper-ului **@model**. Astfel, pentru a afisa informatiile stocate in proprietatile unui model, se utilizeaza urmatoarea secventa de cod:

```
@model Curs7.Models.Student
```

Includerea modelului necesar pentru afisare

```
@{
    ViewBag.Title = "Afisare student";
}

<h1>@Model.Name</h1>
<p>@Model.Email</p>
<p>@Model.CNP</p>
```

Se utilizeaza @Model pentru preluarea si afisarea datelor

Pentru a putea trimite Modelul catre View si pentru a putea fi folosit este nevoie de urmatoarea secenta de cod in Controller:

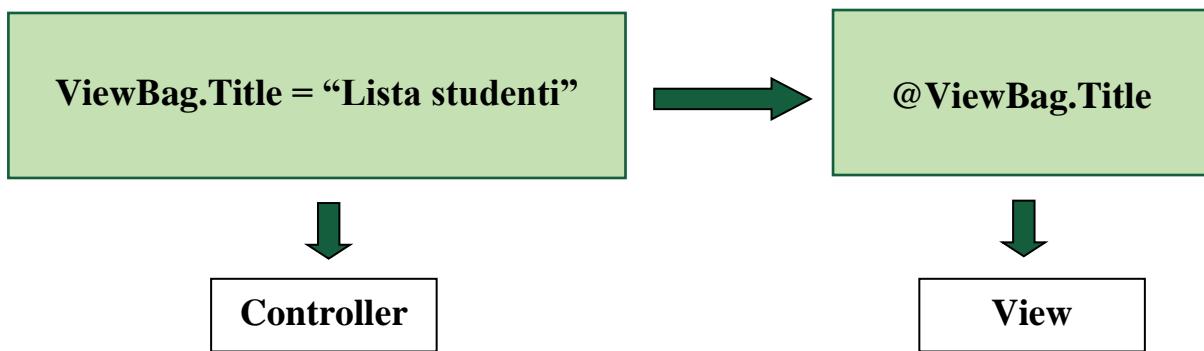
```
public IActionResult Show(int id)
{
    Student student = db.Students.Find(id);
    return View(student);
}
```

Aceasta secenta de cod paseaza obiectul de tip **Model** (in exemplul acesta, un obiect de tipul clasei **Student**) catre View.

## ViewBag

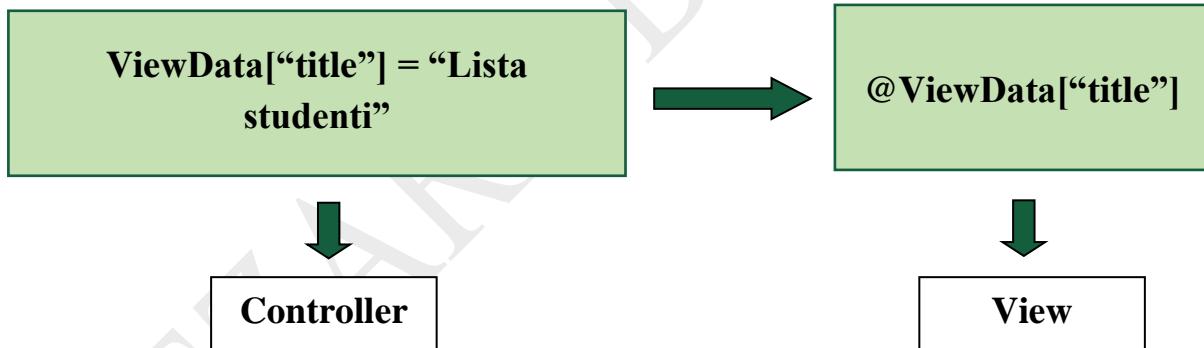
**Helperul ViewBag** – ofera posibilitatea transferului de date intre Controller si View. Aceste date sunt cele care nu se regasesc in Model.

Array-urile care contin obiecte de tipul unui Model, se pot trimite catre View tot prin ViewBag.



## ViewData

**Helperul ViewData** – este similar cu ViewBag, singura diferenta dintre acestea fiind ca **ViewData** este reprezentat de un Dictionar si nu de un obiect, similar cu un array cheie-valoare. Fiecare cheie trebuie sa fie de tip string.



## ⚠️ OBSERVATIE

**ViewBag** insereaza datele alocate proprietatilor in dictionarul asociat variabilei **ViewData**. Acest lucru necesita ca numele cheilor (pentru ViewData) respectiv numele proprietatilor (pentru ViewBag) sa fie **diferite**.

### Exemplu:

ViewBag.Title = "Titlu";  
 ViewData["Title"] = "Titlu 2";

Acest lucru conduce la suprascrierea valorii atributului "Title" din ViewBag (adica Titlu), cu ultima valoare alocata in cheia din ViewData (si anume "Titlu 2")

Astfel, in View, atat @ViewBag.Title, cat si @ViewData["Title"] vor afisa "Titlu 2".

### TempData

**Helperul TempData** – poate seta o valoare care va fi disponibila intr-un request subsecvent. Astfel, daca valoarea a fost setata in Actiunea1, iar aceasta actiune va face un redirect catre Actiunea2, valoarea setata in TempData va fi disponibila in Actiunea2 (**ATENTIE! Doar la prima accesare**).

**Exemplu:** Sa presupunem ca avem C.R.U.D. pentru obiectul Student. Controller-ul are metoda Delete care va sterge studentul din baza de date, prin verbul HTTP Post, neafisand in acest caz un View. Aceasta metoda executa codul aferent stergerii din baza de date si redirectioneaza catre metoda Index.

Pentru stergerea unui student, se vor executa 2 request-uri HTTP dupa cum urmeaza:

- **View-ul Show** (care afiseaza datele unui student si butonul de stergere) – aceasta este pagina unde are loc request-ul, prin apasarea butonului de stergere. La apasarea butonului se va face un Request catre metoda Delete din Controller
  
- **[Request 1]:** Se va accesa metoda Delete si se va executa codul aferent stergerii din baza de date. Dupa stergere, metoda redirectioneaza catre Index.

- [Request 2]: Browser-ul va ajunge in metoda Index si va prelua lista studentilor, impreuna cu mesajul primit din cadrul primului request, prin intermediul variabilei TempData. Aceste valori sunt apoi trimise in View-ul Index pentru afisarea catre utilizatorul final.

Pentru o mai buna experienta de utilizare a aplicatiei (**UX – User Experience**) este necesara afisarea catre utilizatorul final a unui mesaj. Mesajul are ca scop instiintarea utilizatorului cu privire la actiunea pe care tocmai a executat-o (resursa a fost stearsa cu succes). Acest lucru trebuie sa se intampla in pagina Index si se poate realiza prin intermediul helper-ului **TempData** (deoarece avem doua request-uri subsecvente).

## Implementare:

**View-ul Show** – din acest View o sa se execute request-ul (stergerea unui student din baza de date, in functie de id-ul acestuia)

```
@model Curs7.Models.Student

@{
    ViewBag.Title = "Afisare student";
}

<h1>@Model.Name</h1>
<p>@Model.Email</p>
<p>@Model.CNP</p>

<form method="post" action="/Students/Delete/@Model.Id">
    <button class="btn btn-danger" type="submit">Stergere
    student</button>
</form>
```



**In StudentsController -> metoda Delete** – in aceasta metoda se executa codul aferent **request-ului 1**, si anume se va sterge un student din baza de date. Dupa stergere, metoda redirectioneaza catre Index. In acest request variabila TempData[“message”] o sa stocheze stringul in cheia message, dupa care o sa trimita valoarea catre request-ul 2 → Index.

```
[HttpPost]
public ActionResult Delete(int id)
{
    Student student = db.Students.Find(id);
    TempData["message"] = "Studentul cu numele " +
student.Name + " a fost sters din baza de date";

    db.Students.Remove(student);
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

**In StudentsController -> metoda Index** – browser-ul va ajunge in metoda Index si va prelua lista studentilor, impreuna cu mesajul primit din cadrul primului request, prin intermediul variabilei TempData. Aceste valori sunt apoi trimise in View-ul Index pentru afisarea catre utilizatorul final.

```
public ActionResult Index()
{
    var students = from student in db.Students
                   select student;

    ViewBag.Students = students;

    if (TempData.ContainsKey("message"))
    {
        ViewBag.Msg = TempData["message"].ToString();
    }
    return View();
}
```

**In View-ul Index** – sunt afisati toti studentii din baza de date, dar si mesajul provenit din TempData.

```
<h2>Afisare lista studenti</h2>
<h2>@ViewBag.Msg</h2>
@foreach (var student in ViewBag.Students)
{
    <p>@student.Name</p>
    <p>@student.Email</p>
    <p>@student.CNP</p>
    <br />
    <a class="btn btn-success"
        href="/Students>Show/@student.Id">Afisare student</a>
    <br />
}
```

## /Students/Index

localhost:7029/Students/Index

Curs7 Home Privacy

### Afisare lista studenti

Pop Tudor  
tudor@gmail.com  
1930210456778

Afisare student

Popescu Ana  
ana@gmail.com  
2940823567856

Afisare student

## /Students>Show/id

The screenshot shows a web page with the URL `localhost:7029/Students>Show/4`. At the top, there are navigation icons and a link to the current page. Below the header, there is a horizontal menu with links for `Curs7`, `Home`, and `Privacy`. The main content area displays the following information:

**Pop Tudor**  
**tudor@gmail.com**  
**1930210456778**

**Stergere student**

## /Students/Index

The screenshot shows a web page with the URL `localhost:7029/Students`. At the top, there are navigation icons and a link to the current page. Below the header, there is a horizontal menu with links for `Curs7`, `Home`, and `Privacy`. The main content area displays the following message:

**Afisare lista studenti**  
**Studentul cu numele Pop Tudor a fost sters din baza de date**

Popescu Ana  
ana@gmail.com  
2940823567856

**Afisare student**

## Helpere pentru View

ASP.NET MVC Core, prin intermediul motorului Razor ofera o lista de **Helpere** care pot genera elemente de tip HTML. Aceste Helpere sunt folosite in combinatie cu un Model pentru a usura munca dezvoltatorului in generarea formularelor de procesare a datelor acestuia.

Aceste **Helpere utilizeaza sistemul de binding** pentru a afisa valorile modelului in elementele de tip HTML (**de exemplu**: in cazul editarii datelor unui model) cat si pentru trimitera acestora catre Controller. Toate Helperele se acceseaza prin intermediul obiectului **@Html** disponibil in View.

Printre cele mai importante **Helpere** se enumera:

- **Html.ActionLink** – genereaza un URL
- **Html.TextBox** – genereaza un element de tipul TextBox
- **Html.TextArea** – genereaza un element de tipul TextArea
- **Html.CheckBox** – genereaza un element de tipul Check-box, util pentru valorile de tip Boolean
- **Html.RadioBox** – genereaza un element de tipul Radio button
- **Html.DropDownList** – genereaza un element de tipul Dropdown, util pentru valorile de tip Enum
- **Html.ListBox** – genereaza un element de tipul Dropdown cu selectie multipla
- **Html.Hidden** – genereaza un input field ascuns
- **Html.Password** – genereaza un camp pentru introducerea parolelor (textul introdus in camp este ascuns)
- **Html.Display** – este util pentru afisarea textelor
- **Html.Label** – genereaza un label pentru un element mentionat anterior
- **Html.Editor** – acest helper genereaza unul din elementele de mai sus in functie de tipul proprietatii modelului. Astfel, daca editorul este alocat unui camp de tip **int** va genera un input de tip numeric; daca editorul este alocat unui camp de tip string va genera un textbox, etc.

Folosirea acestor Helpere este similara cu scrierea manuala a codului HTML aferent formularelor, insa, Helperele **ofera si posibilitatea de binding a datelor in mod automat**.

**Exemplu:** sa consideram formularul urmator pentru **adaugarea unui student in baza de date**.

```
<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 />
    <button type="submit">Adauga student</button>
</form>
```

Acesta se va rescrie, folosind **Helperele Html** dupa cum urmeaza:

```
<form method="post" action="/Students/New">
    @Html.Label("Name", "Nume Student")
    <br />

    @Html.TextBox("Name", null, new { @class = "form-control" })
    <br /><br />

    @Html.Label("Email", "Adresa de e-mail")
    <br />

    @Html.TextBox("Email", null, new { @class = "form-control" })
    <br /><br />

    @Html.Label("CNP", "CNP Student")
    <br />

    @Html.TextBox("CNP", null, new { @class = "form-control" })
    <br />

    <button type="submit">Adauga student</button>
</form>
```

Generarea unui label pentru atributul "Name"

Generarea unui input field pentru proprietatea Email

Folosind aceste Helpere, codul HTML generat de View este urmatorul:

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

Pentru **label** se genereaza urmatoarele elemente:

```
@Html.Label("Name", "Nume Student")
<label for="Name">Nume Student</label>
```

Pentru **TextBox** se genereaza urmatoarele elemente:

```
@Html.TextBox("Name", null, new { @class = "form-control" })
<input id="Name" name="Name" type="text" value="" class="form-
control" />
```

Continutul HTML generat de Helpere este similar cu cel scris manual. Insa, in cazul **editarii, unde este necesar sa preluam valorile existente in Model** pentru modificarea ulterioara a acestora, putem apela la helperul **Html.Editor** pentru a genera in mod automat campurile de editare si pentru a prelua automat aceste valori.

Formularul initial pentru **editarea unui student**, implementat manual cu preluarea manuala a valorilor si alocarea acestora inputurilor HTML prin intermediul atributului **value** are urmatorul continut:

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

    <label>Nume</label>

    <input type="text" name="Name" value="@ViewBag.Student.Name" />

    <label>Adresa e-mail</label>

    <input type="text" name="Email" value="@ViewBag.Student.Email" />

    <label>CPN</label>
    <input type="text" name="CNP" value="@ViewBag.Student.CNP" />

    <button type="submit">Modifica student</button>

</form>
```

Acesta poate fi rescris prin intermediul helperelor astfel:

```
@model Curs7.Models.Student

<form method="post" action="/Students/Edit/@Model.Id">

    @Html.Label("Name", "Nume Student")
    <br />
    @Html.Editor("Name")
    <br /><br />

    @Html.Label("Email", "Adresa de e-mail")
    <br />
    @Html.Editor("Email") ← Acest helper genereaza in mod automat campul necesar
    <br /><br />

    @Html.Label("CNP", "CNP Student")
    <br />
    @Html.Editor("CNP")
    <br /><br />

    <button type="submit">Modifica student</button>

</form>
```

## Edit

**Nume Student**

**Adresa de e-mail**

**CNP Student**

Codul generat este urmatorul:

```

<form method="post" action="/Students/Edit/1">

    <label for="Name">Nume Student</label>

    <br />

    <input class="text-box single-line" id="Name" name="Name"
type="text" value="Student 2" />

    <br /><br />

    <label for="Email">Adresa de e-mail</label>

    <br />

    <input class="text-box single-line" data-val="true" data-
val-required="The Email field is required." id="Email"
name="Email" type="text" value="user@test.com" />

    <br /><br />

    <label for="CNP">CNP Student</label>

    <br />

    <input class="text-box single-line" data-val="true" data-
val-maxlength="The field CNP must be a string or array type with
a maximum length of '13'." data-val-maxlength-max="13"
data-val-minlength="The field CNP must be a string or array type
with a minimum length of '13'." data-val-minlength-
min="13" id="CNP" name="CNP" type="text" value="1121123123123" />

    <br /><br />

    <button type="submit">Modifica student</button>

</form>

```

Putem observa ca **Helper-ul Html.Editor** a generat toate campurile necesare pentru formularul de editare a studentului, conform definitiei modelului Student. De asemenea, in formular se observa ca toate atributele **value** asociate elementelor formularului au primit valorile modelului in mod automat (**prin Binding**).

## Implementarea alternativa a Helperelor

**Html.TextBox** – genereaza un element de tipul TextBox → Devine input in noua versiune de utilizare a helperelor. In continuare sunt prezentate ambele variante:

- **Varianta 1 → Helper @Html**

```
@Html.TextBoxFor(m => m.Title, null, new { @class = "form-control" })
```

- **Varianta 2 → Helper in-line**

```
<input asp-for="Title" class="form-control" />
```

**Html.DropDownList** – genereaza un element de tipul Dropdown → Devine select

- **Varianta 1 → Helper @Html**

```
@Html.DropDownListFor(m => m.CategoryId, new SelectList(Model.Categories, "Value", "Text"), "Selectati categoria", new { @class = "form-control" })
```

## ➤ Varianta 2 → Helper in-line

```
<select class="form-control" asp-for="CategoryId" asp-items="Model.Categories">
    <option disabled selected value="">Selectati
    Categoria</option>
</select>
```

**Html.Label** → Devine Label

## ➤ Varianta 1 → Helper @Html

```
@Html.LabelFor(m => m.Title, "Titlu Articol")
```

## ➤ Varianta 2 → Helper in-line

```
<label asp-for="Title"></label>
```

Fiecare Helper are un atribut numit **asp-for** folosit pentru a indica proprietatea pe care modelul o utilizeaza pentru procesul de model binding.

Helperele pot avea si alte atribute care incep cu **asp-** si care sunt folosite pentru diverse procese, cum ar fi:

- popularea unui select cu o lista de optiuni (**asp-items**)
- setarea unei rute pentru un form **asp-controller** si **asp-action**
- setarea unui url pentru un tag : **asp-controller** si **asp-action**

# Dezvoltarea Aplicatiilor Web utilizand ASP.NET Core MVC

## Laborator 7

---

### EXERCITII:

Se considera baza de date, cu cele trei modele Article.cs, Category.cs si Comment.cs din Laborator 6. Modificati atat View-urile, cat si metodele din Controllere (atunci cand este necesar) astfel incat sa se utilizeze helpere pentru View. Pentru exemple – **VEZI Curs 7 Sectiunea Helpere pentru View.**

1. Studiati sectiunea **Trimiterea datelor catre View → Model** din cadrul Cursului 7, dupa care implementati urmatoarea functionalitate:
  - a. Sa se modifice actiunea (metoda) **Show** din Controller-ul Articles, astfel incat sa se trimita Modelul de tip Article catre View-ul asociat. Pasul urmator este modificarea View-ului, astfel incat sa includa modelul si sa afiseze proprietatile acestuia catre utilizatorul final.
  
2. Modificati functionalitatea de **adaugare a unui nou articol** in baza de date, impreuna cu selectarea categoriei din care face parte, astfel:
  - a. **Metoda New cu Get** o sa apeleze o metoda, cu ajutorul careia se vor prelua categoriile din baza de date, in vederea trimiterii acestora in View-ul asociat pentru a popula un element de tip dropdown. Este necesar un dropdown deoarece in momentul in care utilizatorul final doreste sa adauge un nou articol in baza de date, acesta trebuie sa aleaga si o categorie din care o sa faca parte articoul respectiv. Alegerea categoriei o sa se realizeze cu ajutorul un dropdown. **Ce fel de metoda trebuie sa fie metoda prin care se preiau toate categoriile din baza de date?**

```

public IEnumerable<SelectListItem> GetAllCategories()
{
    // generam o lista de tipul SelectListItem fara elemente
    var selectList = new List<SelectListItem>();

    // extragem toate categoriile din baza de date
    var categories = from cat in db.Categories
                     select cat;

    // iteram prin categorii
    foreach (var category in categories)
    {
        // adaugam in lista elementele necesare pentru
        // id-ul categoriei si denumirea acesteia
        selectList.Add(new SelectListItem
        {
            Value = category.Id.ToString(),
            Text = category.CategoryName.ToString()
        });
    }

    /* Sau se poate implementa astfel:
     *
     * foreach (var category in categories)
     {
         var listItem = new SelectListItem();
         listItem.Value = category.Id.ToString();
         listItem.Text = category.CategoryName.ToString();

         selectList.Add(listItem);
     }*/
}

// returnam lista de categorii
return selectList;
}

```

## Implementare dropdown:

Dupa implementarea metodei anterioare, prin care sunt preluate toate categoriile din baza de date, urmeaza apelarea acesteia in metoda New din Controller-ul Articles si stocarea categoriilor cu ajutorul unei proprietati, dupa cum urmeaza:

```
public IActionResult New()
{
    Article article = new Article();
    article.Categ = GetAllCategories();
    return View(article);
}
```

- Se instantiaza clasa Article deoarece in cadrul formularului o sa se creeze un nou obiect in baza de date de tipul modelului Article;
- Se apeleaza metoda `GetAllCategories()` prin care sunt preluate toate categoriile din baza de date si se stocheaza aceste categorii pentru a fi trimise catre View-ul New;
- Stocarea listei care contine toate categoriile din baza de date (de forma id\_categorie – denumire categorie) se realizeaza prin adaugarea unei noi proprietati in Modelul Article:

```
[NotMapped]
public IEnumerable<SelectListItem> Categ { get; set; }
```

Cu ajutorul acestui atribut se pot prelua categoriile in cadrul Helper-ului `@Html.DropDownListFor` astfel:

```
@Html.DropDownList(m => m.CategoryId, new
SelectList(Model.Categ, "Value", "Text"), "Selectati
categoria", new { @class = "form-control" })
```

- Se trimitе un obiect de tipul modelului Article catre View-ul asociat, astfel incat View-ul sa primeasca acest model prin intermediul Helperului **@model**

Folosind Helper-ul **@model** in View pentru modelul Article, putem adauga **@Html.DropDownListFor** pentru acest model. Acest Helper (DropDownListFor) primeste urmatorii parametri:

- Primul parametru este o lambda expresie prin care se alege atributul modelului pentru care se va genera elementul de tip select (in acest caz pentru id-ul categoriei → **CategoryId**);
- Al doilea parametru trebuie sa fie o lista de tipul **SelectList** cu elementele pentru care se va genera acest dropdown
- Al treilea element reprezinta o optiune default (Selectati categoria)
- Al patrulea element este optional si reprezinta o lista de atribute aditionale pentru acest element generat

**View-ul New – implementare utilizand Helpere (@Html.Label, @Html.TextBox, @Html.TextArea, @Html.DropDownListFor):**

```
@model ArticlesApp.Models.Article

<h2 class="text-center mt-5">Adaugare articol</h2>
<br />
<div class="container mt-5">
    <div class="row">
        <div class="col-6 offset-3">

            <form method="post" action="/Articles/New">

                @Html.Label("Title", "Titlu Articol")
                <br />

                @Html.TextBox("Title", null, new { @class = "form-control" })

            </form>
        </div>
    </div>
</div>
```

```

<br /><br />

@Html.Label("Content", "Continut Articol")
<br />
@Html.TextArea("Content", null, new { @class = "form-
control" })
<br /><br />

<label>Selectati categoria</label>
@Html.DropDownList(m => m.CategoryId, new
SelectList(Model.Categ, "Value", "Text"),
"Selectati categoria", new { @class = "form-control"
})
<br />

<button class="btn btn-success" type="submit">Adauga
articol</button>

</form>
</div>
</div>
</div>

```

- b. **Metoda New cu POST** – sa se adauge articolul in baza de date si sa se afiseze un mesaj sugestiv “Articolul a fost adaugat”. Pentru afisarea mesajului o sa se utilizeze o variabila de tip TempData conform **Curs 7 – Sectiunea Trimiterea Datelor catre View -> TempData**.

Ce trebuie sa returnam in cazul in care adaugarea articolului in baza de date nu s-a putut realiza cu succes (pe ramura catch())?

3. Modificati functionalitatea de **editare a unui articol existent** in baza de date, impreuna cu selectarea categoriei din care face parte, astfel:
  - a. Actiunea Edit cu GET sa foloseasca metoda `GetAllCategories()` pentru preluarea categoriilor din baza de date si popularea unui element de tip `@Html.DropDownListFor`
  - b. Actiunea Edit cu GET o sa trimita catre View-ul asociat un model de tip Article
  - c. View-ul Edit trebuie sa contine Helpere pentru View (`@Html.Label, @Html.EditorFor, @Html.DropDownListFor`)

```

@model ArticlesApp.Models.Article

<form method="post" action="/Articles/Edit/@Model.Id">

    @Html.Label("Title", "Titlu Articol")
    <br />
    @Html.EditorFor(m => m.Title, new { htmlAttributes =
new { @class = "form-control" } })
    <br /><br />
    @Html.Label("Content", "Continut Articol")
    <br />
    @Html.Editor("Content", new { htmlAttributes = new {
@class = "form-control" } })
    <br /><br />
    @Html.HiddenFor(m => m.Date)
    <label>Selectati categoria</label>
    @Html.DropDownListFor(m => m.CategoryId,
new SelectList(Model.Categ, "Value", "Text"),
"Selectati categoria", new { @class = "form-control" })
    <br />
    <button class="btn btn-sm btn-success"
type="submit">Modifica articol</button>

</form>

```

- d. Actiunea Edit cu POST sa realizeze adaugarea modificarilor in baza de date si sa se afiseze un mesaj corespunzator: “Articolul a fost modificat”, utilizand o variabila de tip TempData.
  
  
- 4. Modificati functionalitatea de stergere a unui articol din baza de date, astfel incat in metoda Delete, dupa realizarea operatiei de stergere, sa se afiseze mesajul: “Articolul a fost sters”, utilizand o variabila de tip TempData.
  
  
- 5. Sa se procedeze la fel ca in implementarile anterioare si pentru entitatea Category (utilizarea helperelor pentru View si a mesajelor de tip TempData pentru afisarea mesajelor sugestive catre utilizatorul final).

# Dezvoltarea Aplicatiilor Web utilizand ASP.NET Core MVC

## Curs 8

---

### Cuprins

View (interfata cu utilizatorul).....	2
Validarea cu ajutorul atributelor .....	2
Atribute de validare la nivel de Model .....	2
Error Message .....	3
Exemple de implementare la nivel de Model .....	3
Preluarea validatorilor in View .....	9
Exemplu de implementare la nivel de View .....	10
Helper-ul @Html.BeginForm .....	11
@Html.ValidationMessage si @Html.ValidationMessageFor .....	12
Helper-ul @Html.ValidationSummary .....	12
Vizualizarea mesajelor de validare in browser .....	13
Implementarea validatorilor la nivel de Controller.....	16
View-uri partajate .....	16
Layout View.....	17
Adaugarea unui nou Layout.....	19
Partial View.....	26

# View (interfata cu utilizatorul)

## Validarea cu ajutorul atributelor

In ASP.NET Core MVC validarea se poate realiza prin intermediul adaugarii atributelor necesare in Model. Atributele de validare ofera posibilitatea integrarrii regulilor de validare pentru fiecare atribut/proprietate a modelului.

### Atribute de validare la nivel de Model

Cele mai utilizate atribute de validare sunt urmatoarele:

- **Required** – verifica daca inputul este diferit de null;
- **StringLength** – verifica daca lungimea unui string este mai mica sau egala decat limita impusa;
- **Range** – verifica daca valoarea inputului se afla intr-un anumit interval;
- **RegularExpression** – verifica daca valoarea respecta expresia regulata;
- **CreditCard** – verifica daca valoarea are un format specific unui cod bancar;
- **CustomValidation** – reprezinta o validare custom (creata de dezvoltator pentru a valida un anumit atribut);
- **EmailAddress** – verifica daca valoarea inputului are un format specific unei adrese de e-mail;
- **FileExtension** – verifica daca valoarea corespunde unei denumiri de extensie;

- **MaxLength** – specifica valoarea maxima pe care o poate avea un array sau un string;
- **MinLength** – specifica valoarea minima pe care o poate avea un array sau un string;
- **Phone** – verifica daca valoarea reprezinta un numar de telefon valid;
- **DataType** – verifica tipul de date al inputului;

## Error Message

Atributele de validare permit utilizarea parametrului **ErrorMessage** pentru afisarea catre utilizatorul final a unui mesaj de eroare specific, in functie de validarea utilizata pentru respectivul atribut.

### Exemplu:

```
[Required(ErrorMessage = "Continutul articolului este obligatoriu")]
```

## Exemple de implementare la nivel de Model

### Exemplul 1 – validari asupra modelului Student

Se adauga asupra modelului Student urmatoarele validari:

- **Numele** este obligatoriu
- **Emailul** este obligatoriu si trebuie sa aiba un format specific unei adrese de e-mail
- **CNP-ul** este obligatoriu si trebuie sa aiba exact 13 caractere
- **Adresa** este obligatorie si nu poate avea mai mult de 30 de caractere

```

public class Student
{
    [Key]
    public int StudentId { get; set; }

    [Required(ErrorMessage = "Numele studentului este
obligatoriu")]
    public string Name { get; set; }

    [Required(ErrorMessage = "Campul e-mail este
obligatoriu")]
    [EmailAddress(ErrorMessage = "Adresa de e-mail nu este
valida")]
    public string Email { get; set; }

    [MinLength(13, ErrorMessage = "Lungimea minima trebuie
sa fie de 13 caractere")]
    [MaxLength (13, ErrorMessage = "Lungimea maxima trebuie
sa fie de 13 caractere")]
    [Required(ErrorMessage = "CNP-ul este obligatoriu")]
    public string CNP { get; set; }

    [StringLength(50, ErrorMessage = "Adresa nu poate avea
mai mult de 50 de caractere")]
    [Required(ErrorMessage = "Adresa este obligatorie")]
    public string Address { get; set; }
}

```

**Exemplul 2 – validari asupra modelului Article (exemplul din cadrul laboratorului)**

Se adauga asupra modelului Article urmatoarele validari:

- **Titlul** articolului este obligatoriu, poate avea o lungime maxima de 100 de caractere si nu poate avea mai putin de 5 caractere
- **Continutul** articolului este obligatoriu
- **Categoria** din care face parte articolul este obligatorie

```

public class Article
{
    [Key]
    public int Id { get; set; }

    [Required(ErrorMessage = "Titlul este obligatoriu")]
    [StringLength(100, ErrorMessage = "Titlul nu poate avea
mai mult de 100 de caractere")]
    [MinLength(5, ErrorMessage = "Titlul trebuie sa aiba mai
mult de 5 caractere")]
    public string Title { get; set; }

    [Required(ErrorMessage = "Continutul articolului este
obligatoriu")]
    public string Content { get; set; }

    public DateTime Date { get; set; }

    [Required(ErrorMessage = "Categoria este obligatorie")]
    public int CategoryId { get; set; }

    public virtual Category Category { get; set; }

    public virtual ICollection<Comment> Comments { get; set; }

    [NotMapped]
    public IEnumerable<SelectListItem> Categ { get; set; }
}

```

## **!OBSERVATII**

Mesajele de eroare aflate in Model, ca valoare a parametrului ErrorMessage, o sa fie preluate in View pentru afisare.

Pentru preluarea mesajelor de validare in View **VEZI Sectiunea urmatoare din curs – Preluarea Validatorilor in View**. Dupa adaugarea validatorilor in Model, urmeaza preluarea mesajelor de validare in View.

In cazul in care in formularul de adaugare a unui nou articol nu se completeaza toate inputurile, o sa apara mesajele de eroare asociate, deoarece **Title**, **Content** si **CategoryId** sunt campuri obligatorii. In cazul in care titlul are o lungime mai mica de 5 caractere, atunci o sa apara mesajul de eroare asociat.

De asemenea, se poate observa urmatoarea problema in cazul in care se incerca adaugarea unui articol fara completarea campurilor:

## Adaugare articol

Titlu Articol  
  
 Titlul este obligatoriu

Continut Articol  
  
 Continutul articolului este obligatoriu

Selectati categoria  
  
 Selectati categoria  
 The value " is invalid.

**Adauga articol**

Pentru **Titlu** si **Continut** mesajele de eroare se afiseaza corect. In cazul ultimului input, dropdown-ul in care se selecteaza categoria, mesajul de eroare nu este cel scris in Modelul Article.

Se intampla acest lucru deoarece **CategoryId** este cheie externa. Cheia externa poate fi si null, acest lucru ducand la prezenta acelui mesaj de validare, mesaj implementat by default. Chiar daca acest atribut este obligatoriu, in implementarea lui interna se considera ca poate fi si null. In momentul in care este null, se afiseaza automat mesajul de validare intern “The value is invalid”.

In acest caz, este nevoie ca acel camp sa fie optional la nivel de Model, astfel incat in momentul in care inputul nu are o valoare selectata sa preia corect mesajul configurat la nivel de Model si sa paseze acel mesaj pentru afisare in View.

In acelasi mod trebuie implementate si proprietatile aflate in Modelul Article deoarece proprietatea Category, de exemplu, nu primeste o valoare exact in momentul in care pentru un articol se selecteaza categoria din care face parte. Acest pas are loc dupa asocierea unui articol cu o categorie. Asadar, si in cazul proprietatilor este nevoie ca acestea sa fie declarate *optional* la nivel de Model.

### Codul devine:

```
...
[Required(ErrorMessage = "Categoria este obligatorie")]
public int? CategoryId { get; set; }

public virtual Category? Category { get; set; }

public virtual ICollection<Comment>? Comments {get; set;}

[NotMapped]
public IEnumerable<SelectListItem>? Categ { get; set; }
...
```

## Adaugare articol

Titlu Articol

Titlul este obligatoriu

Continut Articol

Continutul articolului este obligatoriu

Selectati categoria

Selectati categoria

Categoria este obligatorie

**Adauga articol**

In cazul in care Titlul este completat, dar are mai putin de 5 caractere, o sa se afiseze mesajul urmator de eroare, conform validarii.

## Adaugare articol

Titlu Articol

Titlul trebuie sa aiba mai mult de 5 caractere



Continut Articol

Continutul articolului este obligatoriu

Selectati categoria

Categoria este obligatorie

**Adauga articol**

## Preluarea validarilor in View

Pentru preluarea validarilor si afisarea mesajelor asociate in View, se utilizeaza Helper-ul `@Html.ValidationMessageFor` astfel:

- Primul parametru este o lambda expresie care **selecteaza atributul modelului** pentru care se va afisa mesajul de validare
- Al doilea parametru este un string si reprezinta mesajul de validare afisat pe ecran

In cazul in care acesta este gol sau null se va afisa **mesajul de validare aflat in Model** ca parametru al variabilei ErrorMessage.

In cazul in care acest parametru are o valoare, atunci o sa fie suprascris mesajul din Model cu cel primit ca parametru in cadrul Helper-ului.

In cazul in care in Model nu exista un mesaj configurat pentru o anumita validare, iar in View parametrul este null, atunci se va afisa un mesaj default, generat de framework.

- Al treilea parametru este optional si reprezinta o lista de atribute care poate fi adaugata mesajului afisat

## Exemplu de implementare la nivel de View

```

@using (Html.BeginForm(actionName: "New", controllerName:
"Articles"))
{
    @Html.Label("Title", "Titlu Articol")
    <br />
    @Html.TextBox("Title", null, new { @class = "form-control" })
}

@Html.ValidationMessageFor(m => m.Title, null, new {
@class = "text-danger" })
<br /><br />

@Html.Label("Content", "Continut Articol")
<br />
@Html.TextArea("Content", null, new { @class = "form-
control" })

@Html.ValidationMessage("Content", null, new { @class =
"text-danger" })
<br /><br />

<label>Selectati categoria</label>
@Html.DropDownListFor(m => m.CategoryId, new
SelectList(Model.Categ, "Value", "Text"),
"Selectati categoria", new { @class = "form-control" })

@Html.ValidationMessageFor(m => m.CategoryId, null, new {
@class = "text-danger" })
<br /><br />

<button class="btn btn-success" type="submit">Adauga
articol</button>
}

```

## Helper-ul @Html.BeginForm

Se poate observa utilizarea unui nou Helper → `Html.BeginForm`

Acest Helper inlocuieste tagul `<form>` utilizat pana in acest moment in cazul formularelor din View.

Implementarea unui formular utilizand Helper-ul:

```
@using (Html.BeginForm (actionName: "NumeMetoda",
controllerName: "NumeController",
method: FormMethod.NumeVerbHttp,
routeValues: new { id = Model.NumeId }))
```

In cazul in care se utilizeaza Helper-ul si este nevoie de trimiterea unui id catre Controller (de exemplu in cazul editarii) exista urmatoarele variante de implementare:

1. Trimiterea id-ului prin intermediul Helper-ului `Html.BeginForm` cu ajutorul parametrului `routeValues`, unde Model este Modelul primit ca parametru din Controller si inclus in View prin intermediul Helper-ului `@model` → `@model NumeProject.Models.NumeClasa` (**EX din laborator:** `@model ArticlesApp.Models.Article`)

```
@using (Html.BeginForm(actionName: "Edit", controllerName:
"Articles", routeValues: new { id = Model.Id }))
```

2. Trimiterea id-ului in interiorul formularului prin intermediul Helper-ului `@Html.HiddenFor`

`@Html.HiddenFor(m => m.Id)` -> unde m este un parametru de tipul Modelului primit ca parametru din Controller si inclus in View la fel ca in exemplul anterior

## @Html.ValidationMessage si @Html.ValidationMessageFor

### **!OBSERVATIE**

Implementarile urmatoare sunt echivalente, acest lucru fiind valabil in cazul tuturor Helperelor:

```
@Html.ValidationMessageFor(m => m.Title, null, new { @class = "text-danger" })
```

si

```
@Html.ValidationMessage("Title", null, new { @class = "text-danger" })
```

## Helper-ul @Html.ValidationSummary

Helperul `@Html.ValidationSummary` ofera posibilitatea afisarii unui **sumar cu toate erorile aparute in timpul validarii**.

Acesta se adauga in formularul din cadrul View-ului:

```
@Html.ValidationSummary(false, "", new { @class = "text-danger" })
```

Pentru afisarea corecta a mesajelor de eroare, doar in momentul in care validarea datelor nu este corecta, este necesar sa adaugam urmatoarele linii de cod in fisierul **site.css** aflat in wwwroot → folderul css:

```
.field-validation-valid {  
    display: none;  
}  
.validation-summary-valid {  
    display: none;  
}
```

Este necesar sa procedam asa doar in momentul in care mesajele de validare se afla doar in View, in cadrul parametrului doi.

```
@Html.ValidationMessageFor(m => m.Title, "Titlul este obligatoriu", new { @class = "text-danger" })
```

## Vizualizarea mesajelor de validare in browser

Se pot observa mesajele de validare preluate din cadrul Modelului Article, atat la nivelul fiecarui input prin intermediul Helper-ului `@Html.ValidationMessageFor`, dar si mesajele preluate prin intermediul Helper-ului `@Html.ValidationSummary`

## Adaugare articol

• Titlul este obligatoriu  
 • Continutul articolului este obligatoriu  
 • Categoria este obligatorie



`@Html.ValidationSummary`

Titlu Articol	Titlul este obligatoriu
Continut Articol	Continutul articolului este obligatoriu
Selectati categoria	Selectati categoria
	Categoria este obligatorie

**Adauga articol**

In cazul in care exista un mesaj de validare in View, acesta o sa suprascrie mesajul configurat la nivel de Model, dupa cum urmeaza:

### Mesajul de validare din View-ul New:

```
@Html.TextBox("Title", null, new { @class = "form-control" })
@Html.ValidationMessageFor(m => m.Title, "OBLIGATORIU", new
{ @class = "text-danger" })
```

### Mesajul de validare din Modelul Article:

```
[Required(ErrorMessage = "Titlul este obligatoriu")]
public string Title { get; set; }
```

Se poate observa faptul ca mesajul de validare din View il suprascrie pe cel din Model doar la nivelul inputului. În cazul în care se realizează sumașul cu toate mesajele de validare, sunt preluate tot mesajele de validare existente la nivel de Model.

De aceea este recomandată utilizarea mesajelor de validare în Model.

În cazul în care sunt utilizate validări, fără scrierea mesajelor specifice, framework-ul o să afiseze mesaje default.

**De exemplu:** “The Title field is required.”

“The field Title must be a string or array type with a minimum length of '5'.”

## Adaugare articol

- Titlul este obligatoriu
- Continutul articolului este obligatoriu
- Categorie este obligatorie

Titlu Articol

OBLIGATORIU

Continut Articol

Continutul articolului este obligatoriu

Selectați categoria

Selectați categoria

Categorie este obligatorie

**Adauga articol**

## Implementarea validatorilor la nivel de Controller

Pentru functionarea corecta a validatorilor, cat si pentru identificarea corecta a datelor in partea de server (server-side) este necesar sa adaugam in Controller-ul care modifica datele, verificarea starii modelului. Astfel, prin intermediul variabilei **ModelState** putem sa aflam daca toate validatorile au trecut cu succes.

```
[HttpPost]
public IActionResult New(Article article)
{
    article.Date = DateTime.Now;
    article.Categ = GetAllCategories();

    if (ModelState.IsValid)
    {
        db.Articles.Add(article);
        db.SaveChanges();
        TempData["message"] = "Articolul a fost
adaugat";
        return RedirectToAction("Index");
    }
    else
    {
        return View(article);
    }
}
```

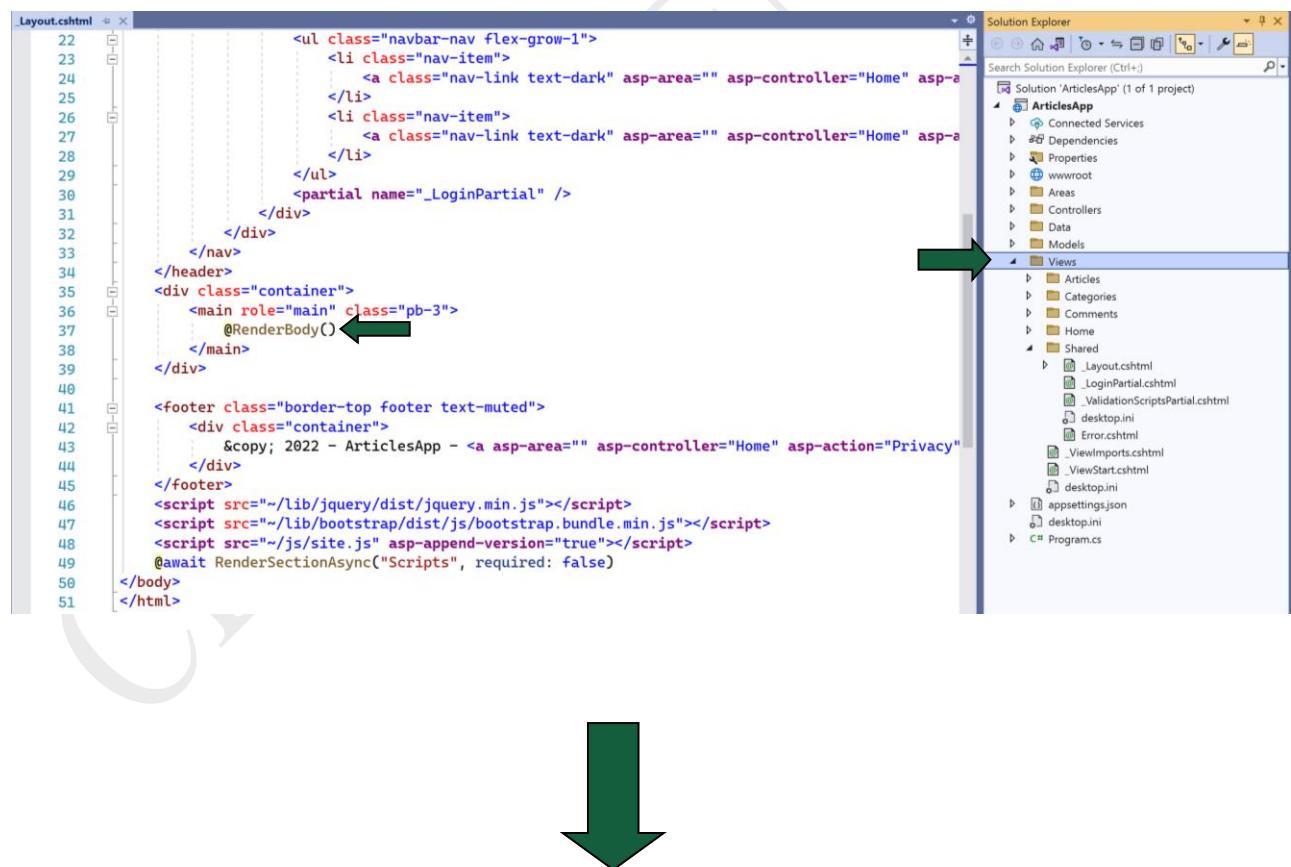
## View-uri partajate

Interfata unei aplicatii, indiferent de tehnologia cu care este realizata, intotdeauna o sa contine foarte multe componente comune tuturor paginilor: Header, Footer, Meniuri, etc. Aceste componente nu se modifica de la o pagina la alta, iar repetarea scrierii aceliasi cod devinde redundanta. Pentru a facilita implementarea se pot utiliza View-uri globale.

## Layout View

**Layout View** – permite scrierea unui cod comun pentru toate paginile, cat si un Placeholder in care se va include continutul celorlalte pagini. Acest placeholder este definit prin intermediul variabilei **@RenderBody()**. Locul in care este plasata aceasta variabila in Layout, va fi locul in care se va afisa continutul View-urilor aferente.

De exemplu, in momentul in care cream un nou proiect, acesta genereaza in mod automat un layout care include toate resursele necesare: Head, Stiluri CSS, JavaScript, Header, Footer, etc. In acest layout se afla metoda **RenderBody()** prin care toate View-urile create sunt incluse.



Solution Explorer

Solution 'ArticlesApp' (1 of 1 project)

- ArticlesApp
  - Connected Services
  - Dependencies
  - Properties
  - wwwroot
  - Areas
  - Controllers
  - Data
  - Models
  - Views
    - Articles
    - Categories
    - Comments
    - Home
    - Shared
      - \_Layout.cshtml
      - \_LoginPartial.cshtml
      - \_ValidationScriptsPartial.cshtml
      - desktop.ini
      - Error.cshtml
      - \_ViewImports.cshtml
      - \_ViewStart.cshtml
      - desktop.ini
    - appsettings.json
    - desktop.ini
  - C# Program.cs

**Layout.cshtml**

```

22 <ul class="navbar-nav flex-grow-1">
23   <li class="nav-item">
24     <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
25   </li>
26   <li class="nav-item">
27     <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
28   </li>
29 </ul>
30 <partial name="_LoginPartial" />
31 </div>
32 </div>
33 </header>
34 <div class="container">
35   <main role="main" class="pb-3">
36     @RenderBody()
37   </main>
38 </div>
39 </div>
40
41 <footer class="border-top footer text-muted">
42   <div class="container">
43     &copy; 2022 - ArticlesApp - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
44   </div>
45 </footer>
46 <script src="~/lib/jquery/dist/jquery.min.js"></script>
47 <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
48 <script src="~/js/site.js" asp-append-version="true"></script>
49 @await RenderSectionAsync("Scripts", required: false)
50 </body>
51 </html>

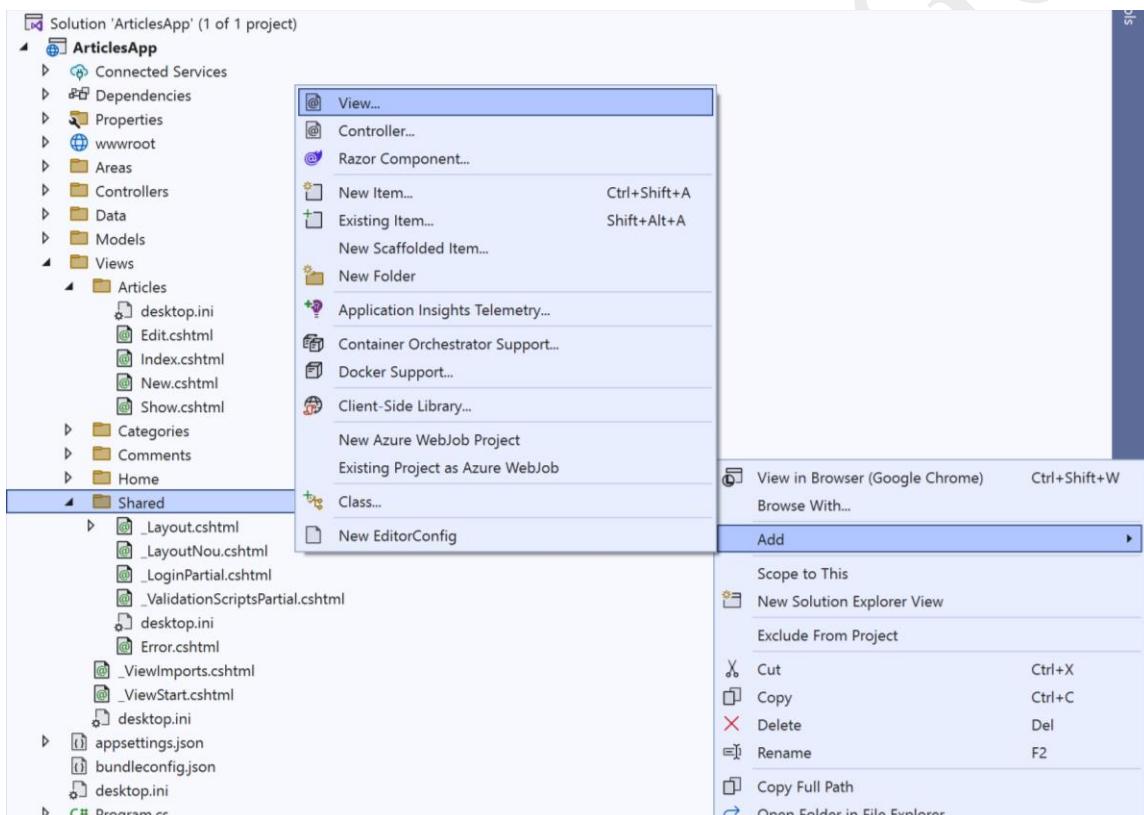
```

## Adaugarea unui nou Layout

Implicit, proiectul ASP.NET Core are in folderul Shared din View un Layout. Acest Layout este utilizat in cadrul tuturor paginilor existente in proiect. In momentul in care se doreste utilizarea unui alt Layout, se poate crea si se utilizeaza dupa cum urmeaza:

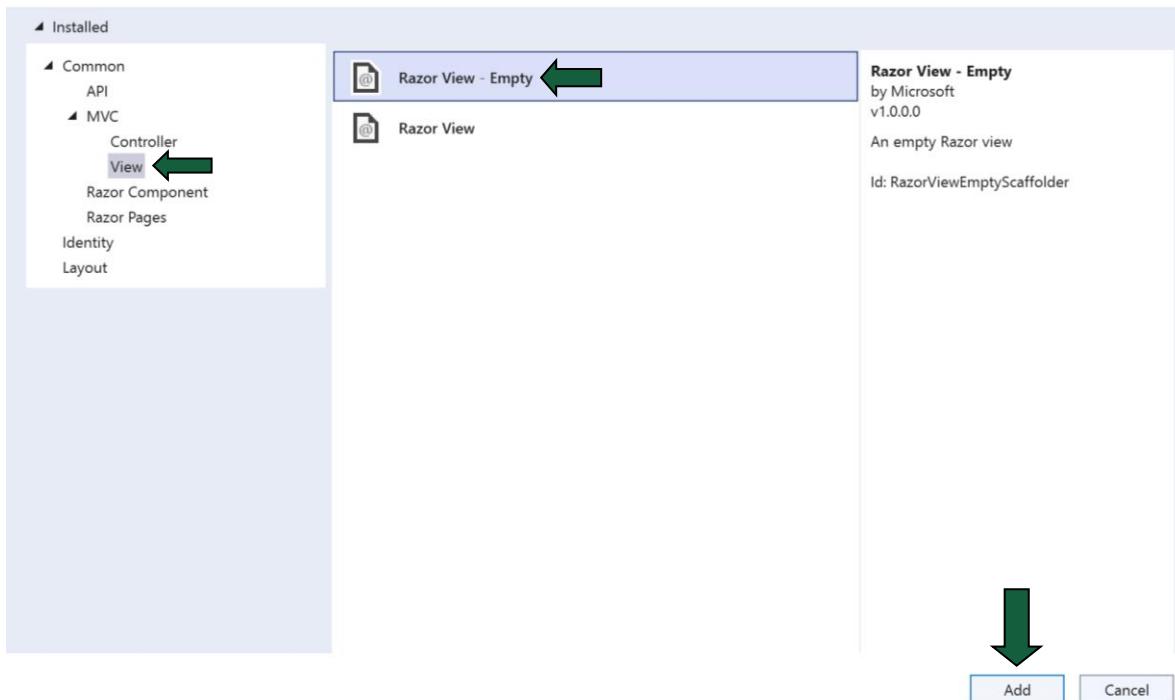
### PASUL 1 – crearea noului Layout

Views → Shared → click dreapta → Add → View

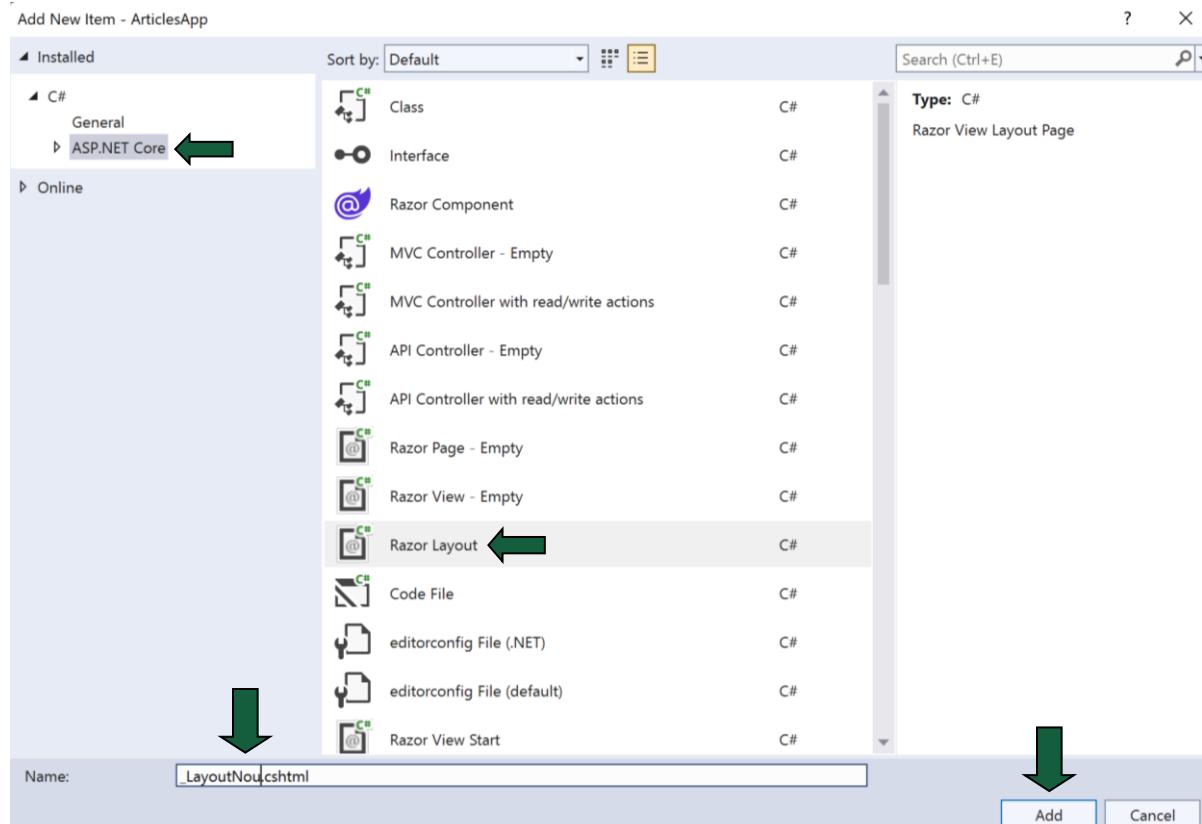


## Se selecteaza Razor View – Empty

Add New Scaffolded Item



**Se selecteaza ASP.NET Core → Razor Layout**  
**Se redenumeste Layout-ul → Add**



**View-ul generat de framework o sa arate astfel:**

```
_LayoutNou.cshtml
1  <!DOCTYPE html>
2
3  <html>
4  <head>
5      <meta name="viewport" content="width=device-width" />
6      <title>@ViewBag.Title</title>
7  </head>
8  <body>
9      <div>
10         @RenderBody()
11     </div>
12  </body>
13  </html>
```

## PASUL 2 – modificarea noului Layout, adaugand header si footer

The screenshot shows a code editor with two tabs: '\_LayoutNou.cshtml' and 'IndexNou.cshtml'. The '\_LayoutNou.cshtml' tab is active, displaying the following code:

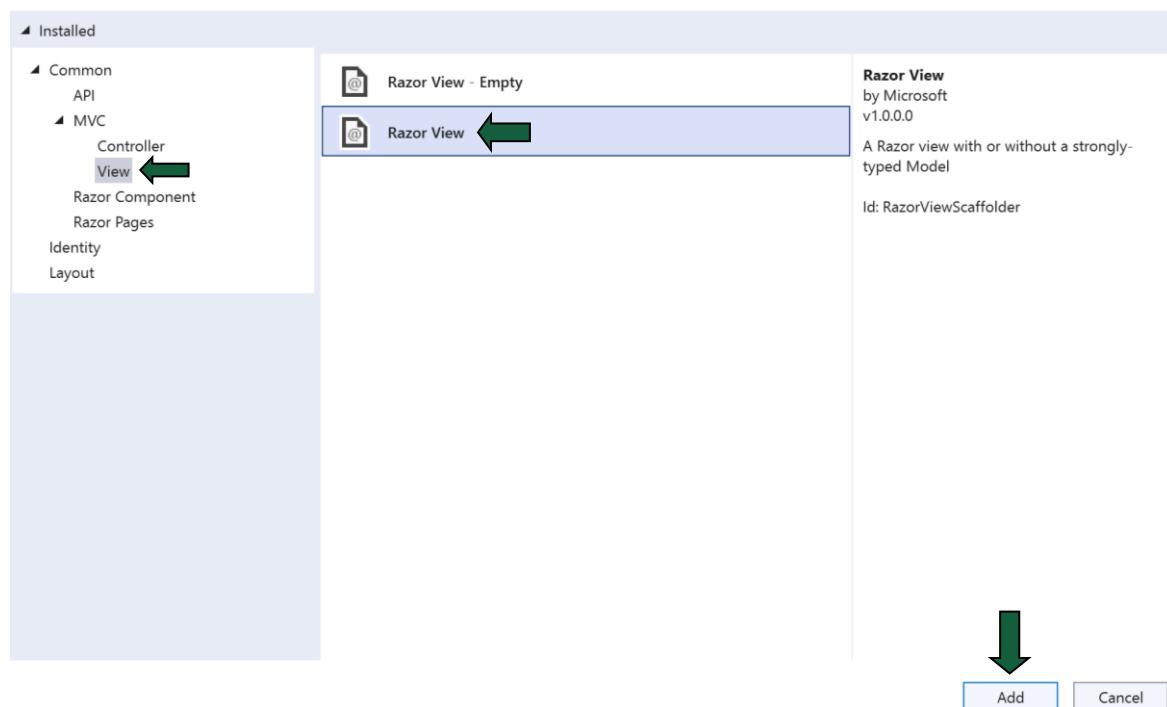
```
13 <header>
14   <div class="container">
15     <div class="col-md-6 offset-3 mt-5">
16       <h2>Header</h2>
17     </div>
18   </div>
19 </header>
20
21 <body>
22   <div class="container">
23     <div class="col-md-6 offset-3">
24       @RenderBody()
25     </div>
26   </div>
27 </body>
28
29 <footer>
30   <div class="container">
31     <div class="col-md-6 offset-3">
32       <h2>Footer</h2>
33     </div>
34   </div>
35 </footer>
36 </html>
```

The 'IndexNou.cshtml' tab is visible but contains no code.

## PASUL 3 – adaugarea unui nou View care o sa aiba ca Layout noul Layout creat:

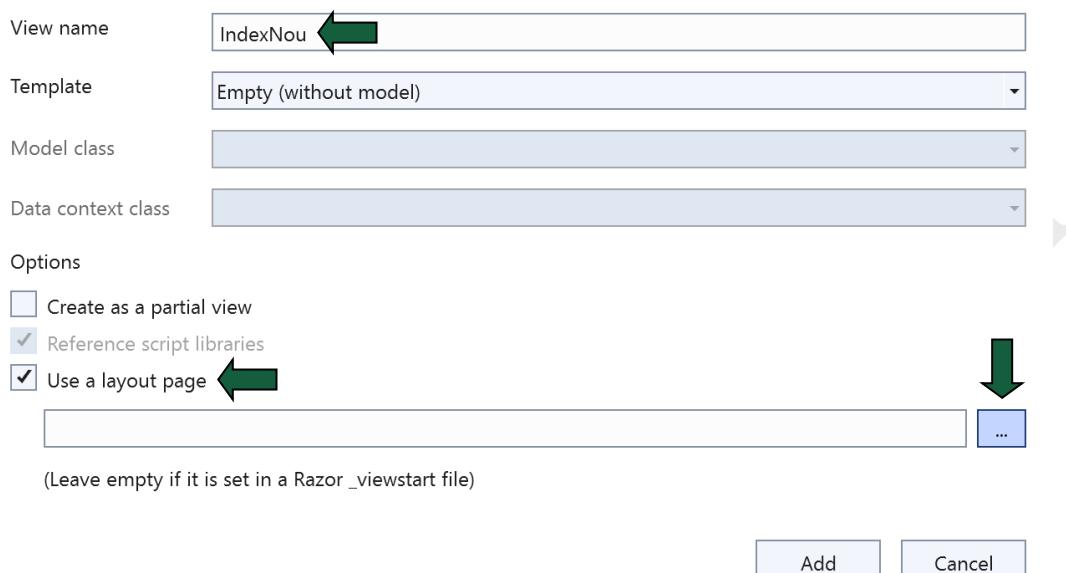
Se adauga un nou View de tipul Razor View

Add New Scaffolded Item

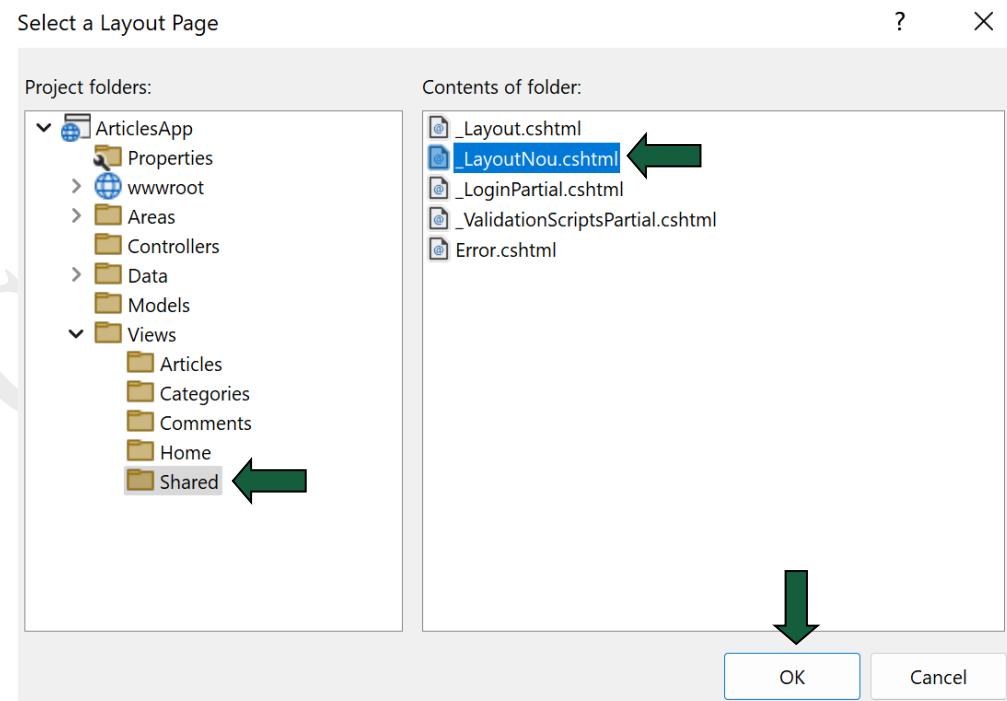


**Se adauga un nume pentru respectivul View, dupa care se alege Layout-ul**

### Add Razor View

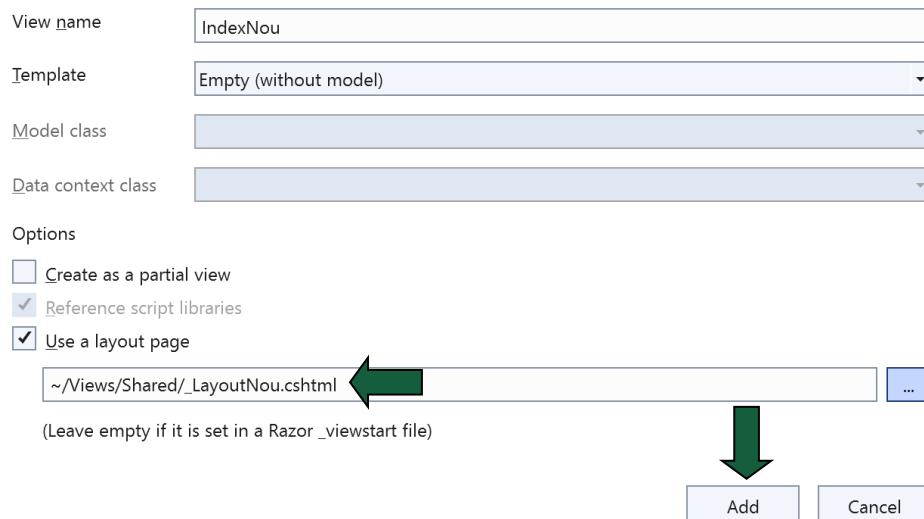


### Se selecteaza Layout-ul



## In acest pas se observa Layout-ul pe care tocmai l-am selectat

### Add Razor View



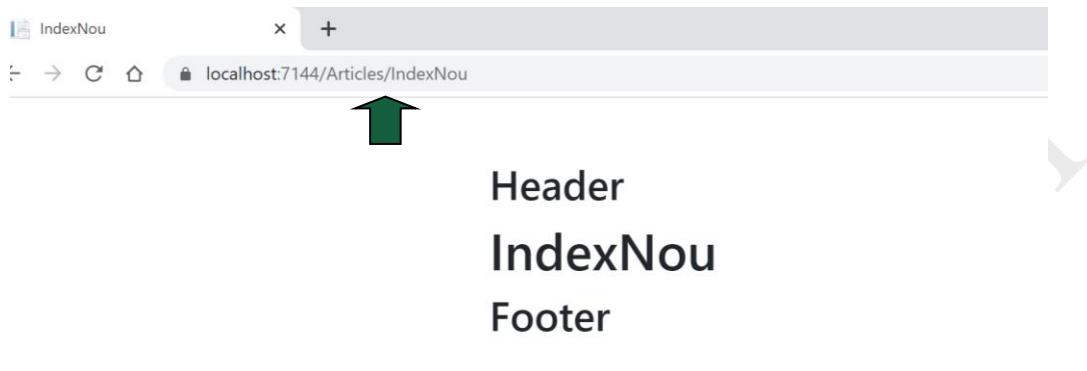
Dupa executarea pasilor anteriori, se poate observa cum in cadrul View-ului numit *IndexNou*, Layout-ul numit *LayoutNou* se include prin intermediul parametrului *Layout* care primeste ca valoare calea din sistemul de fisiere. Mai jos se poate observa sevenita de cod generata:

```

IndexNou.cshtml _Layout.cshtml _LayoutNou.cshtml
1  @{
2      ViewData["Title"] = "IndexNou";
3      Layout = "~/Views/Shared/_LayoutNou.cshtml";
4  }
5
6
7  <h1>IndexNou</h1>
8
9

```

Dupa rulare si accesarea URL-ului → /Articles/IndexNou se poate observa includerea cu succes a noului Layout (noul Layout nu are stilizare, exemplul fiind realizat cu scop demonstrativ).



## Partial View

**Partialele** reprezinta secente de cod specific View-urilor care pot fi reutilizate in una sau mai multe pagini. In cadrul dezvoltarii aplicatiilor web, codul poate fi reutilizat pentru a optimiza timpul de scriere si pentru a nu include acelasi cod in mod repetitiv.

Secente de cod care se repeta in cadrul mai multor pagini pot fi incluse intr-un View sau intr-un Layout pentru a fi afisate.

In cadrul aplicatiei dezvoltate in laborator, afisarea unui articol contine acelasi cod, atat in cazul afisarii tuturor articolelor din baza de date → View-ul Index, cat si in cazul afisarii unui singur articol → View-ul Show. In acest caz, pentru a elimina redundanta, si anume scrierea repetitiva a aceluiasi cod in cadrul ambelor View-uri, o sa utilizam un **Partial View**, dupa cum urmeaza:

Secventa de cod, cu ajutorul careia se afiseaza informatiile corespunzatoare unui articol, se afla inclusa in ambele View-uri, *Index.cshtml* si *Show.cshtml*.

```
<div class="card-body">

    <h3 class="card-title alert-success py-3 px-3 rounded-2">@Model.Title</h3>

    <div class="card-text">@Model.Content</div>

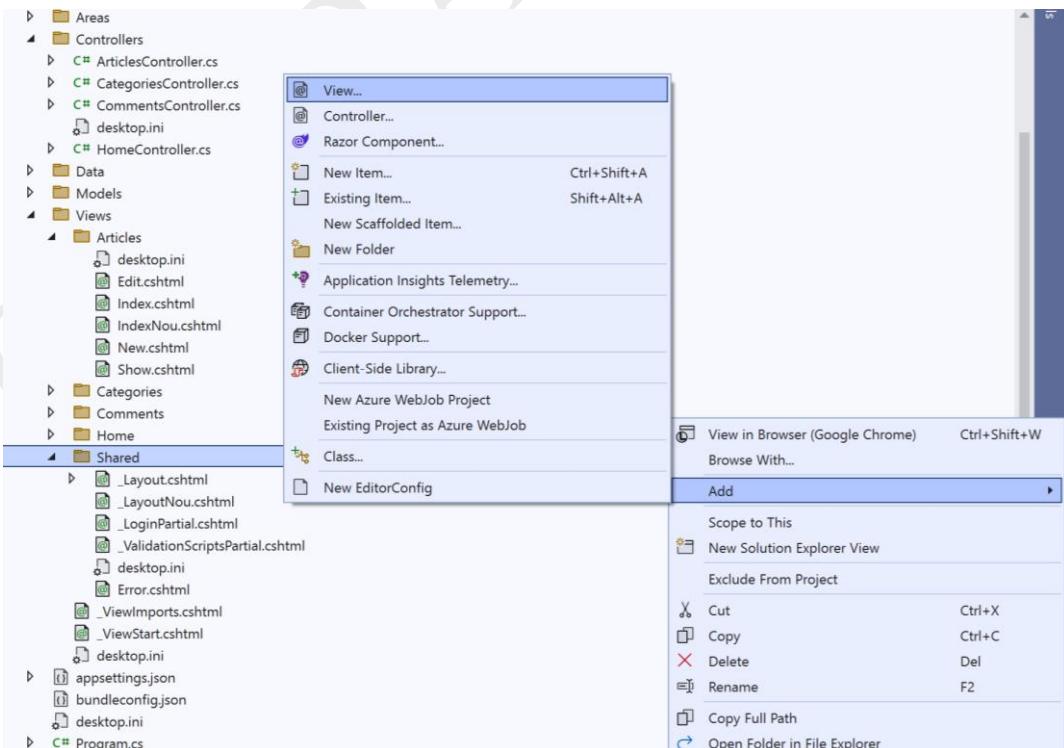
    <div class="d-flex justify-content-between flex-row mt-5">

        <div><i class="bi bi-globe"></i>
        @Model.Category.CategoryName</div>

        <span class="alert-success">@Model.Date</span>

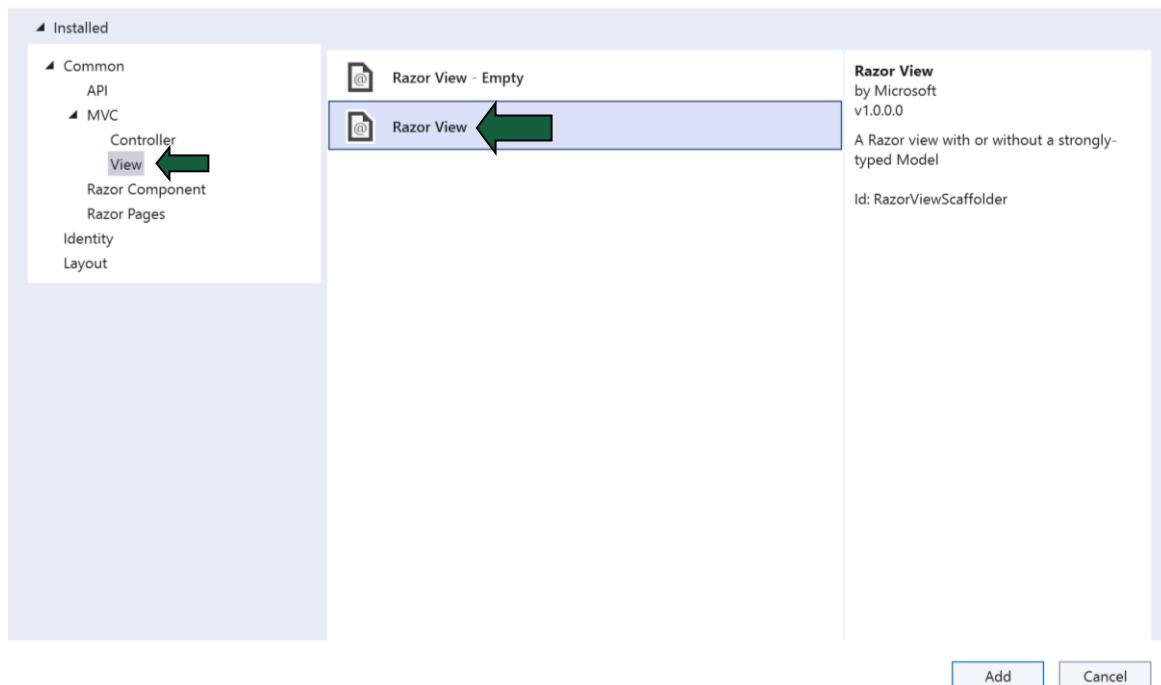
    </div>
</div>
```

Pentru eliminarea redundantei, se adauga un nou View de tip Partial View. Acest View o sa contine secventa de cod care se repeat.



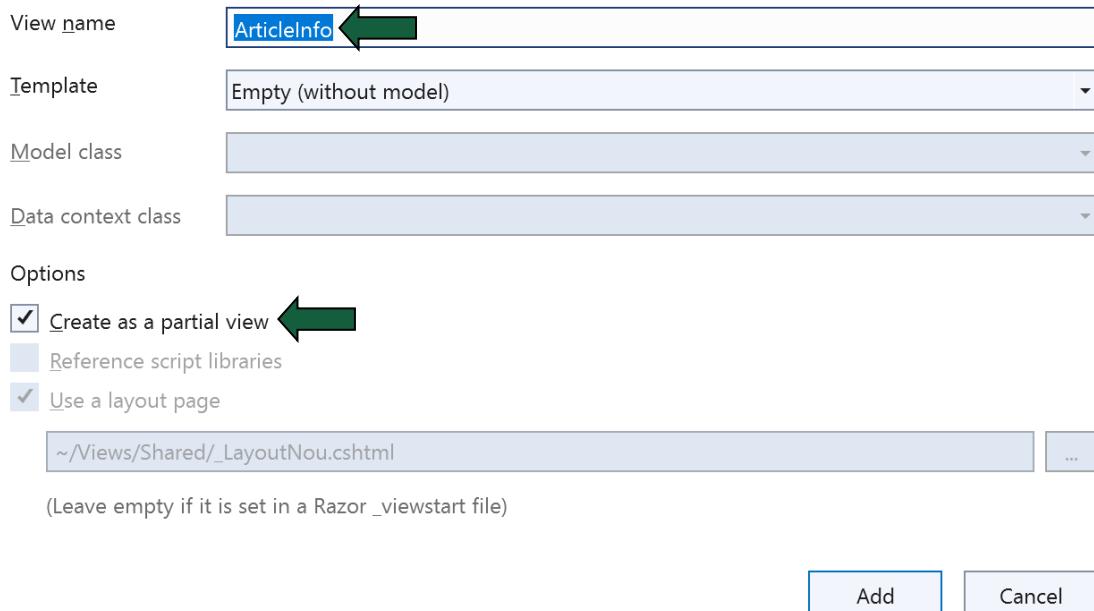
Se selecteaza urmatoarele optiuni:

### Add New Scaffolded Item



View-ul o sa fie de tipul Partial View, bifand optiunea  
 “Create as a partial view”

## Add Razor View



View-ul numit “ArticleInfo” o sa contina sevenita de cod care se repeta in ambele View-uri, atat in Index.cshtml, cat si in Show.cshtml.

```

1
2
3
4 <div class="card-body">
5   <h3 class="card-title alert-success py-3 px-3 rounded-2">@Model.Title</h3>
6   <div class="card-text">@Model.Content</div>
7
8   <div class="d-flex justify-content-between flex-row mt-5">
9     <div><i class="bi bi-globe"></i> @Model.Category.CategoryName</div>
10    <span class="alert-success">@Model.Date</span>
11  </div>
12
13
14
15
16
17
18 </div>

```

Pentru includerea partialului se utilizeaza Helper-ul specific `@Html.Partial` sau tag-ul `<partial></partial>`

### **Exemplu utilizand aplicatia dezvoltata in cadrul laboratorului:**

Pentru apelarea partialului din **View-ul Show** se utilizeaza pentru primul parametru numele partialului, iar pentru al doilea paramentru **Modelul**.

```
@Html.Partial("ArticleInfo", Model)
```

Pentru apelarea partialului din **View-ul Index** se utilizeaza pentru primul parametru numele partialului, iar pentru al doilea paramentru obiectul **article** de tipul **Model**. Astfel, in loop trebuie sa declaram tipul modelului si sa pasam acest parametru la partial. Prin intermediul acestui cod, in partial putem sa folosim variabila `@Model` pentru afisarea datelor.

```
@foreach (ArticlesApp.Models.Article article in
ViewBag.Articles)
{
    ...
    @Html.Partial("ArticleInfo", article)
    ...
}
```

In cazul modificarii partialului, modificarile se reflecta asupra tuturor View-urilor care utilizeaza partialul respectiv, nefiind nevoie de modificari in fiecare View in parte. Acest lucru eficientizeaza atat timpul de scriere a codului, cat si mentenanta ulterioara a acestuia in cazul in care apar modificari in timp.

# Dezvoltarea Aplicatiilor Web utilizand ASP.NET Core MVC

## Laborator 8

### EXERCITII:

Se considera baza de date, cu cele trei modele Article.cs si Category.cs, Comment.cs, din laboratorul anterior.

Sa se modifice implementarea, acolo unde este necesar, astfel incat sa fie posibila integrarea validarilor necesare. **Cititi cu atentie notiunile din cadrul Cursului 8 si implementati exercitiile urmatoare.**

#### Sugestii de implementare:

1. Sa se modifice Modelele (Article, Category, Comment) adaugandu-se validarile necesare la nivel de Model astfel (**VEZI Curs 8 – Sectiunea Atribute de validare la nivel de Model**):

- Se adauga asupra modelului **Article** urmatoarele validari:
  - **Titlul** articolului este obligatoriu (Required), poate avea o lungime maxima de 100 de caractere (StringLength) si nu poate avea mai putin de 5 caractere (MinLength)
  - **Continutul** articolului este obligatoriu (Required)
  - **Categoria** din care face parte articolul este obligatorie (Required)
- Se adauga asupra modelului **Category** urmatoarea validare:
  - **Numele** categoriei este obligatoriu (Required)
- Se adauga asupra modelului **Comment** urmatoarea validare:
  - **Continutul** comentariului este obligatoriu (Required)

2. Preluati validarile in View-urile asociate (**VEZI Curs 8 – Secțiunea Preluarea validarilor in View**) urmand pasii urmatori. De asemenea, dupa adaugarea validarilor in View, studiați cu atenție **Exemplul 2 din Cursul 8**. In cadrul exemplului se observă de ce mesajele de eroare nu se afisează corect și cum se poate rezolva aceasta problema. Implementați corect mesajele de eroare în cadrul aplicației *ArticlesApp*.
  - Se preiau validările pentru entitatea Article, View-urile New și Edit (adaugarea unui nou articol și editarea unui articol existent)
  - Validările se includ în View cu ajutorul Helper-ului specific `@Html.ValidationMessageFor` sau `@Html.ValidationMessage` cele două fiind echivalente (**VEZI Curs 8 – Secțiunea @Html.ValidationMessage si @Html.ValidationMessageFor**)
  - Înlocuiți tagul `<form>` utilizat pentru crearea unui formular, cu Helper-ul specific (**VEZI Curs 8 – Secțiunea Helper-ul @Html.BeginForm**)
  - Afisați un sumar cu toate erorile aparute în timpul validării (**VEZI Curs 8 – Secțiunea Helper-ul @Html.ValidationSummary**)
3. Implementați validările la nivel de Controller, astfel încât acestea să se afișeze corect în funcție de acțiunea pe care o face utilizatorul final (**VEZI Curs 8 – Secțiunea Implementarea validarilor la nivel de Controller**)

## Explicatii privind adaugarea validatorilor la nivel de Controller:

In metoda **New** cu **HttpPost** trebuie sa adaugam si verificarea starii modelului. Prin intermediul variabilei **ModelState** se verifica daca toate validatorile au trecut cu succes si se pot salva modificarile in baza de date.

```
[HttpPost]
public IActionResult New(Article article)
{
    article.Date = DateTime.Now;
    article.Categ = GetAllCategories();

    if (ModelState.IsValid)
    {
        db.Articles.Add(article);
        db.SaveChanges();
        TempData["message"] = "Articolul a fost
adaugat";
        return RedirectToAction("Index");
    }
    else
    {
        return View(article);
    }
}
```

In momentul in care validatorile nu trec cu succes, ramura de executie va fi cea din **else**. Astfel, in momentul in care returnam View-ul este necesar sa trimitem din nou modelul impreuna cu toate atributele sale.

### ➤ La fel se procedeaza si pentru editare

```
[HttpPost]
public IActionResult Edit(int id, Article requestArticle)
{
    Article article = db.Articles.Find(id);
    requestArticle.Categ = GetAllCategories();

    if(ModelState.IsValid)
    {
        article.Title = requestArticle.Title;
        article.Content = requestArticle.Content;
        article.CategoryId = requestArticle.CategoryId;
        TempData["message"] = "Articolul a fost modificat";
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    else
    {
        return View(requestArticle);
    }
}
```

4. Pentru o mentenanta mai buna a codului si pentru a elimina redundanta, utilizati un Partial View pentru afisarea informatiilor asociate unui articol.

In cadrul aplicatiei, afisarea unui articol contine acelasi cod, atat in cazul afisarii tuturor articolelor din baza de date → View-ul Index, cat si in cazul afisarii unui singur articol → View-ul Show. In acest caz, pentru a elmina redundanta, si anume scrierea repetitiva a aceluiasi cod in cadrul ambelor View-uri, sa se utilizeze un View Partial numit **ArticleInfo** (**VEZI Curs 8 – Sectiunea View-uri partajate – Partial View**).

5. Sa se modifice Layout-ul existent, astfel incat sa contina link-uri catre pagina de afisarea a tuturor articolelor, catre pagina de afisare a tuturor categoriilor si catre pagina de adaugare a unui nou articol. O sa se modifice Layout-ul existent, numit \_Layout.cshtml, aflat in folderul View → folderul Shared. In cadrul acestui Layout o sa se adauge cele trei link-uri. De asemenea, o sa se modifice link-ul existent in Layout, cel care la apasarea logo-ului duce la /Home/Index. Dupa modificare, acesta trebuie sa redirectioneze catre /Articles/Index.

Pentru stilizare se pot modifica clasele existente de css, adaugand urmatoarele sechete de cod in **wwwroot → css → site.css**

Sugestii de stilizare (pentru paleta de culori se poate utiliza:  
<https://flatuic平. com/>)

```
.navbar {
    background-color: #dfe6e9 !important;
    border: none !important;
    box-shadow: 0 3px 12px #636e72 !important;
}

.a.nav-link {
    color: #2c2c2c !important;
}
```

```

.a.nav-link:hover {
    color: #00b894 !important;
}

.logoutbtn {
    color: #2c2c2c !important;
}

.logoutbtn:hover {
    color: #00b894 !important;
}

```

6. Sa se adauge validarile si pentru entitatile **Category** si **Comment**.

- **In Model** – Denumirea categoriei este obligatorie si Continutul comentariului este obligatoriu
- **In View** – sa se utilizeze Helper-ul specific pentru validare
- **In Controller** – sa se verifice starea modelului (se verifica daca validarile au trecut cu succes)

### **⚠️ OBSERVATIE**

In cazul adaugarii unui comentariu, formularul de adaugare se afla in View-ul Show asociat Controller-ului Articles. Am procedat in acest mod deoarece un comentariu este asociat unui articol si este necesara afisarea acestuia impreuna cu articolul de care depinde.

Asadar, in pagina Show a articolului, se afiseaza articolul impreuna cu toate comentariile sale, dar si cu un formular in care utilizatorul poate adauga un nou comentariu articolului respectiv. Implementarea formularului este urmatoarea:

```

<form method="post" action="/Articles/Show/@Model.Id">

    <div class="card-body">

        <input type="hidden" name="ArticleId"
value="@Model.Id" />

        <label>Continut comentariu</label>
        <br />

        <textarea class="form-control"
name="Content"></textarea>

        @Html.ValidationMessage("Content", null, new {
@class = "text-danger"}) 

        <br /><br />

        <button class="btn btn-success "
type="submit">Adauga comentariul</button>

    </div>

</form>

```

In cadrul formularului, am adaugat in acest caz si validarea cu ajutorul Helperului specific.

De asemenea, se poate observa si inputul de tip **hidden** cu ajutorul caruia retinem id-ul articolului caruia ii corespunde comentariul, conform implementarii modelului (se observa cheia externa ArticleId):

```

public class Comment
{
    [Key]
    public int Id { get; set; }

    [Required(ErrorMessage = "Continutul comentariului este
obligatoriu")]
    public string Content { get; set; }

    public DateTime Date { get; set; }

    public int? ArticleId { get; set; } ←

    public virtual Article? Article { get; set; }
}

```

Avand in vedere ca un View nu poate avea mai mult de un Model inclus, nu se pot utiliza Helpere pentru restul componentelor de HTML (input, label, textarea). In acest caz, se pot utiliza in continuare tag-urile de HTML.

Deoarece dorim ca in momentul in care se incearca adaugarea unui comentariu fara continut, utilizatorul sa primeasca mesajul de validare “Comentariul este obligatoriu”, dar in acelasi timp trebuie ca pe ramura *else* sa retrimitem intregul articol impreuna cu toate comentariile pe care le are, trebuie sa procedam astfel:

- O sa mutam metoda New cu `HttpPost` din `CommentsController` in `ArticlesController`
- In `ArticlesController` metoda o sa se numeasca `Show` deoarece dorim ca in momentul in care nu se poate adauga comentariul, sa redireccioneze in metoda `Show` a articolului, astfel incat sa se afiseze corect articolul impreuna cu toate comentariile sale. Se poate proceda in acest mod atunci cand dorim sa pastram obiectul si implicit trebuie sa-l pasam ca argument in View. In cazul nostru, obiectul este de tip `Article` si trebuie pasat ca argument in View-ul `Show` pentru afisarea articolului impreuna cu toate comentariile, dar si cu acel formular in care se poate adauga un nou comentariu. Avand in vedere ca in `ArticlesController` mai avem o metoda `Show`, metoda pe care dorim sa o implementam pentru adaugarea unui nou comentariu se poate numi `Show`, dar are nevoie de un verb `Http` diferit. Adaugarea unui comentariu fiind o actiune de scriere, o sa aiba verbul `HttpPost`.
- Avand in vedere ca in `Show` modelul de baza este `Article`, atunci cand se incearca adaugarea comentariului, se preia si id-ul articolului (id-ul modelului principal din view). Pentru a prelua doar datele comentariului si pentru inserarea corecta a acestora in baza de date, se utilizeaza in metoda `Show` `[FromForm]` astfel incat datele sa fie preluate doar din formular

```
public IActionResult Show([FromForm] Comment comment)
```

```

[HttpPost]
public IActionResult Show([FromForm] Comment comment)
{
    comment.Date = DateTime.Now;

    if (ModelState.IsValid)
    {
        db.Comments.Add(comment);
        db.SaveChanges();
        return Redirect("/Articles/Show/" +
comment.ArticleId);
    }

    else
    {
        Article art =
db.Articles.Include("Category").Include("Comments")
            .Where(art => art.Id ==
comment.ArticleId)
            .First();

        //return Redirect("/Articles/Show/" +
comm.ArticleId);

        return View(art);
    }
}

```

Daca sursa default care trimit datele nu paseaza corect argumentele la nivel de Controller, se pot utiliza urmatoarele atribute pentru a specifica sursa prelucrarii valorilor:

- **[FromQuery]** – preia valorile din query-ul de baze de date
- **[FromRoute]** – preia valorile din ruta
- **[FromForm]** – preia valorile din formular
- **[FromBody]** – prea valorile din corpul request-ului (atunci cand se utilizeaza JSON)
- **[FromHeader]** – preia valorile din headerele Http (fiecare request are headere Http care pot fi interpretate de catre server/aplicatie)

# Dezvoltarea Aplicatiilor Web utilizand ASP.NET Core MVC

## Curs 9

---

### Cuprins

Sistemul de autentificare .....	2
Roluri. Asocierea dintre roluri si utilizatori .....	3
Atributul [Authorize] .....	15
Implementare New, Edit si Delete utilizand roluri .....	15

## Sistemul de autentificare

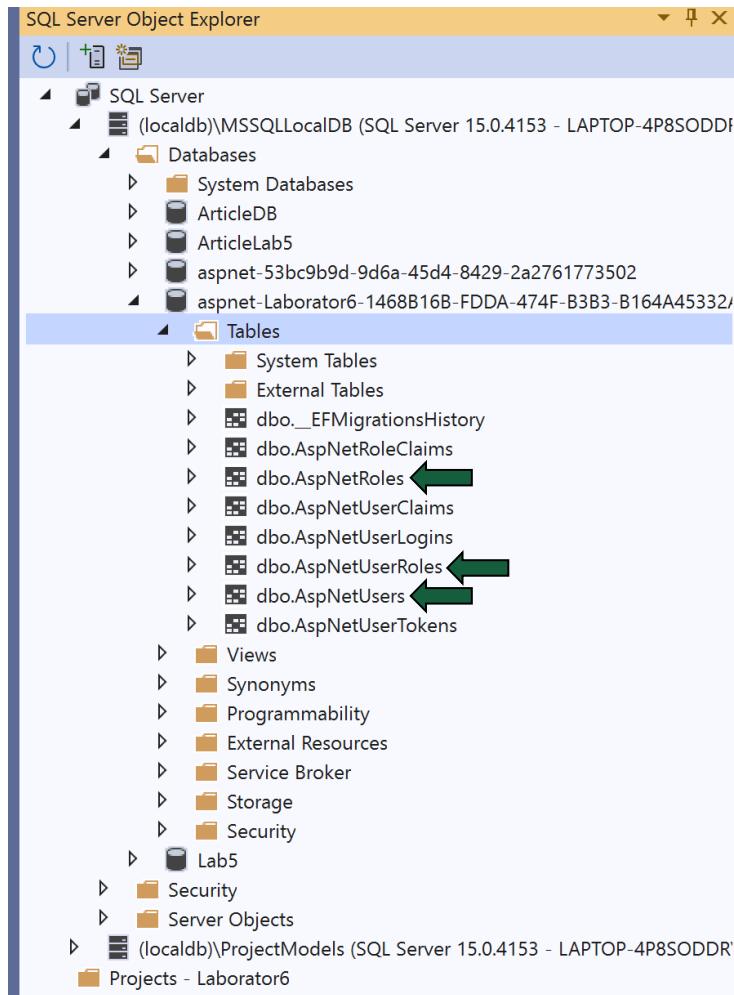
Framework-ul ASP.NET Core ofera posibilitatea integrarii unui sistem de autentificare folosind **Identity**.

**Identity** este compus dintr-o suita de clase si sechete de cod care faciliteaza implementarea rapida a unui sistem de autentificare complex. Acest sistem ofera posibilitatea autentificarii folosind user si parola, alocarea de roluri pentru utilizatori, autentificare folosind conturi 3<sup>rd</sup> party (autentificare prin retele de socializare – Google, Facebook, Twitter, etc). De asemenea, Identity include si posibilitatea crearii si manipularii rolurilor pe care le pot avea utilizatorii. ASP.NET Core Identity utilizeaza baza de date SQL Server pentru stocarea utilizatorilor, a rolurilor, dar si pentru asocierea dintre useri si roluri.

Pentru a genera un proiect care include componenta **Identity** pentru autentificare, trebuie sa alegem la crearea proiectului forma de autentificare: **Individual Accounts**.

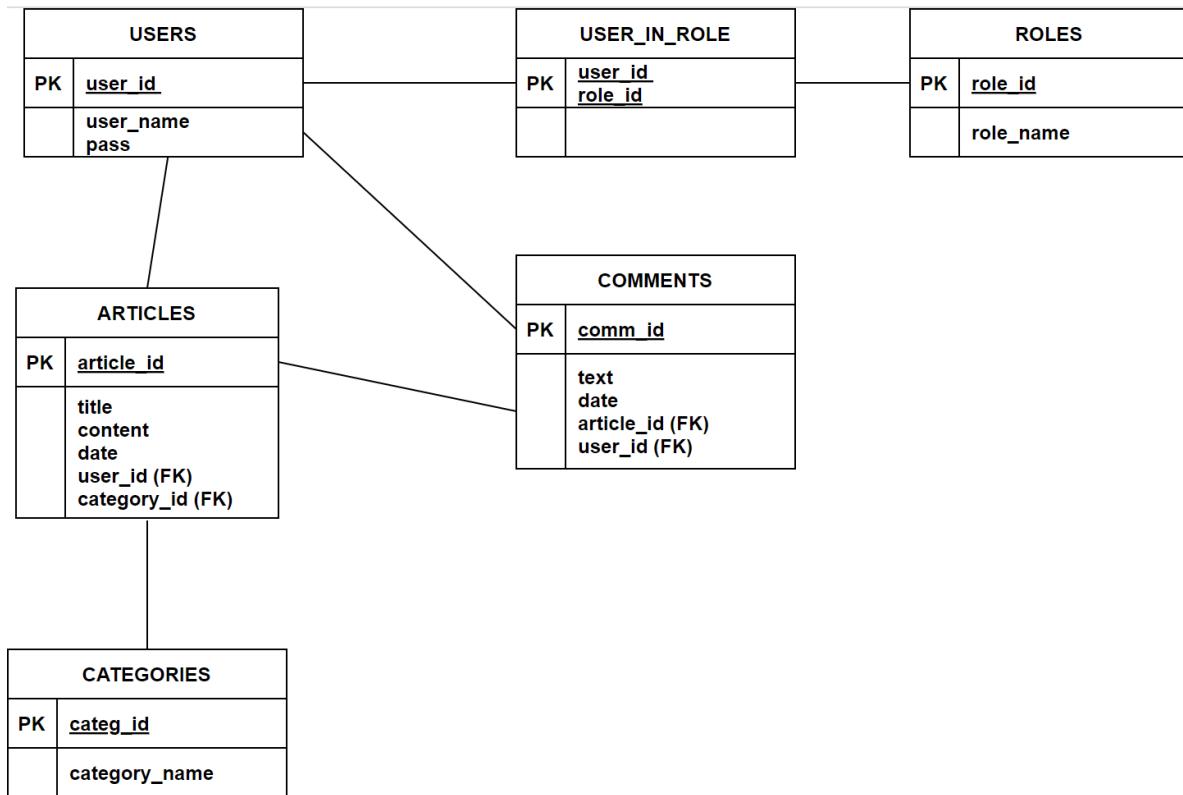
Adaugarea sistemului de autentificare a fost studiata in cadrul **Cursului 6** – **Sectiunea Adaugarea sistemului de autentificare**.

Dupa crearea proiectului impreuna cu sistemul de autentificare, se pot vizualiza tabelele:



## Roluri. Asocierea dintre roluri si utilizatori

Exemplele urmatoare sunt realizate in cadrul aplicatiei implementata in laborator (Engine de stiri), conform urmatoarei diagrame:



Se considera entitatile **Article**, **Category** si **Comment** cu urmatoarele proprietati:

### Article:

- **Id** – int → id-ul articolului (cheie primara)
- **Title** – string → titlul articolului este obligatoriu (Required), poate avea o lungime maxima de 100 de caractere (StringLength) si nu poate avea mai putin de 5 caractere (MinLength)
- **Content** – string → continutul articolului este obligatoriu (Required)
- **Date** – DateTime → data si ora la care este postat articolul
- **Categoria** din care face parte articolul este obligatorie (Required)
- **CategoryId** – int → cheie externa – categoria din care face parte articolul
- **UserId** – string → cheie externa – reprezinta utilizatorul care a postat articolul

## Category:

- **Id** – int → id-ul categoriei (cheie primara)
- **CategoryName** – string → numele categoriei este obligatoriu (Required)

## Comment:

- **Id** – int → id-ul comentariului (cheie primara)
- **Content** – string → continutul comentariului este obligatoriu (Required)
- **Date** – DateTime → data la care a fost postat comentariul
- **ArticleId** – int (cheie externa) → articolul caruia ii apartine comentariul
- **UserId** – string (cheie externa) → utilizatorul care a postat comentariul

In continuare vom modifica anumite configuratii existente si generate de ASP.NET Core Identity, astfel incat sistemul de autentificare sa functioneze si sa putem configura si rolurile din cadrul aplicatiei.

Pentru adaugarea rolurilor si pentru realizarea asocierii dintre utilizatori si roluri trebuie parcursi urmatorii pasi:

### PASUL 1:

Pentru definirea utilizatorilor din aplicatie, Identity include in baza de date un tabel Users, impreuna cu toate atributele necesare (Id, UserName, Email, Password, PhoneNumber, etc).

In cadrul **Pasului 1**, se creeaza o noua clasa in folderul Models, clasa care o sa mosteneasca clasa de baza `IdentityUser` din pachetul `Microsoft.AspNetCore.Identity`. In cazul in care dorim sa extindem clasa `User`, adaugand atribute, putem realiza acest lucru in clasa pe care urmeaza sa o implementam.

Se adauga o clasa pe care o numim **ApplicationUser**

```
namespace ArticlesApp.Models
{
    public class ApplicationUser : IdentityUser
    {

    }
}
```

Clasa pe care o mosteneste, **IdentityUser**, este clasa care descrie Userul in baza de date (cea care contine toate atributele unui utilizator).

## PASUL 2:

In cadrul **Pasului 2** se adauga serviciile necesare in fisierul **Program.cs**

```
builder.Services.AddDefaultIdentity<ApplicationUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<ApplicationContext>();
```

In fisier exista deja configuratia pentru **AddDefaultIdentity** (serviciul de User Interface, cookies si toate mecanismele necesare pentru functionarea autentificarii) si **AddEntityFrameworkStores** (realizeaza conexiunea cu baza de date, conectandu-se la baza de date a carui string de conexiune se afla in appsettings.json), fiind necesara adaugarea serviciului de management al rolurilor → **AddRoles**.

## PASUL 3:

In folderul Data exista contextul bazei de date, fisierul **ApplicationContext.cs**, care contine conexiunea cu baza de date si configurarea provider-ului de baze de date (in cazul nostru SQL Server) folosind dependency injection.

In sevenita urmatoare de cod, clasa ApplicationDbContext mosteneste clasa de baza IdentityDbContext – clasa care se ocupa cu managementul userilor si rolurilor (contine proprietati si metode cu ajutorul carora se pot prelucra userii si rolurile din aplicatie).

Clasa este de forma: `IdentityDbContext<TUser>` ceea ce inseamna ca primeste tipul clasei creata pentru prelucrarea userilor din baza de date.

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options)
        : base(options)
    {
    }
    ...
}
```

#### PASUL 4:

Se creeaza o noua clasa in folderul Models, cu ajutorul careia se vor adauga in baza de date Rolurile necesare. In acelasi timp se pot asocia si utilizatori cu fiecare rol. Acest lucru se face o singura data, urmand ca utilizatorii care creeaza cont sa primeasca un rol in momentul inregistrarii (logica la nivel de Controller).

Se creeaza o clasa, numita sugestiv **SeedData**, in cadrul careia vom implementa o metoda numita **Initialize** prin intermediul careia vom popula cele trei tabele: Users, Roles si UserRoles.

Un Service Provider este utilizat pentru a injecta dependentele in aplicatie (baza de date, sesiuni, pachete, autentificare, etc). In cazul nostru, cerem serviciul care realizeaza conexiunea cu baza de date.

```

public static class SeedData
{
    public static void Initialize(IServiceProvider serviceProvider)
    {
        using (var context = new ApplicationDbContext(
            serviceProvider.GetRequiredService
            <DbContextOptions<ApplicationContext>>()))
        {
            // Verificam daca in baza de date exista cel putin un
            // rol
            // insemanand ca a fost rulat codul
            // De aceea facem return pentru a nu insera rolurile
            // inca o data
            // Aceasta metoda trebuie sa se execute o singura data
            if (context.Roles.Any())
            {
                return; // baza de date contine deja roluri
            }

            // CREAREA ROLURILOR IN BD
            // daca nu contine roluri, acestea se vor crea
            context.Roles.AddRange(
                new IdentityRole { Id = "2c5e174e-3b0e-446f-86af-
483d56fd7210", Name = "Admin", NormalizedName = "Admin".ToUpper() },
                new IdentityRole { Id = "2c5e174e-3b0e-446f-86af-
483d56fd7211", Name = "Editor", NormalizedName = "Editor".ToUpper() },
                new IdentityRole { Id = "2c5e174e-3b0e-446f-86af-
483d56fd7212", Name = "User", NormalizedName = "User".ToUpper() }
            );

            // o noua instanta pe care o vom utiliza pentru
            // crearea parolelor utilizatorilor
            // parolele sunt de tip hash
            var hasher = new PasswordHasher<ApplicationUser>();

            // CREAREA USERILOR IN BD
            // Se creeaza cate un user pentru fiecare rol
            context.Users.AddRange(
                new ApplicationUser
                {
                    Id = "8e445865-a24d-4543-a6c6-9443d048cdb0",
                    // primary key
                    UserName = "admin@test.com",
                    EmailConfirmed = true,
                    NormalizedEmail = "ADMIN@TEST.COM",
                    Email = "admin@test.com",
                    NormalizedUserName = "ADMIN@TEST.COM",
                    PasswordHash = hasher.HashPassword(null,
                    "Admin1!"))
                },
                new ApplicationUser
                {

```

```

        Id = "8e445865-a24d-4543-a6c6-9443d048cdb1",
// primary key
        UserName = "editor@test.com",
        EmailConfirmed = true,
        NormalizedEmail = "EDITOR@TEST.COM",
        Email = "editor@test.com",
        NormalizedUserName = "EDITOR@TEST.COM",
        PasswordHash = hasher.HashPassword(null,
"Editor1!")
    },
    new ApplicationUser
{
    Id = "8e445865-a24d-4543-a6c6-9443d048cdb2",
// primary key
    UserName = "user@test.com",
    EmailConfirmed = true,
    NormalizedEmail = "USER@TEST.COM",
    Email = "user@test.com",
    NormalizedUserName = "USER@TEST.COM",
    PasswordHash = hasher.HashPassword(null,
"User1!")
}
);

// ASOCIEREA USER-ROLE
context.UserRoles.AddRange(
    new IdentityUserRole<string>
    {
        RoleId = "2c5e174e-3b0e-446f-86af-
483d56fd7210",
        UserId = "8e445865-a24d-4543-a6c6-
9443d048cdb0"
    },
    new IdentityUserRole<string>
    {
        RoleId = "2c5e174e-3b0e-446f-86af-
483d56fd7211",
        UserId = "8e445865-a24d-4543-a6c6-
9443d048cdb1"
    },
    new IdentityUserRole<string>
    {
        RoleId = "2c5e174e-3b0e-446f-86af-
483d56fd7212",
        UserId = "8e445865-a24d-4543-a6c6-
9443d048cdb2"
    }
);
context.SaveChanges();
}
}
}

```

Id-urile (Id, RoleId, UserId) au fost generate utilizand:

<https://guidgenerator.com/>

## PASUL 5:

Prin instantia curenta a aplicatiei se apeleaza din clasa SeedData, metoda Initialize, ducand la crearea rolurilor si a utilizatorilor in baza de date.

In **Program.cs** se adauga **Pasul 5**:

```
builder.Services.AddControllersWithViews();

var app = builder.Build();

// PASUL 5 - useri si roluri

using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    SeedData.Initialize(services);
}
```

## PASUL 6:

Se adauga proprietatile in clasele din Model, astfel:

### In clasa Article

```
// PASUL 6 - useri si roluri
public virtual ApplicationUser User { get; set; } → un
articol apartine unui singur utilizator
```

## In clasa Comment

```
// PASUL 6 - useri si roluri
    public virtual ApplicationUser User { get; set; } → un
comentariu apartine unui singur utilizator
```

## PASUL 7:

In folderul Views → Shared → \_LoginPartial.cshtml → se modifica astfel:

```
@inject SignInManager<IdentityUser> SignInManager
@Inject UserManager<IdentityUser> UserManager
```



```
@inject SignInManager<ApplicationUser> SignInManager
@Inject UserManager<ApplicationUser> UserManager
```

## PASUL 8:

Se realizeaza o noua migratie in baza de date.

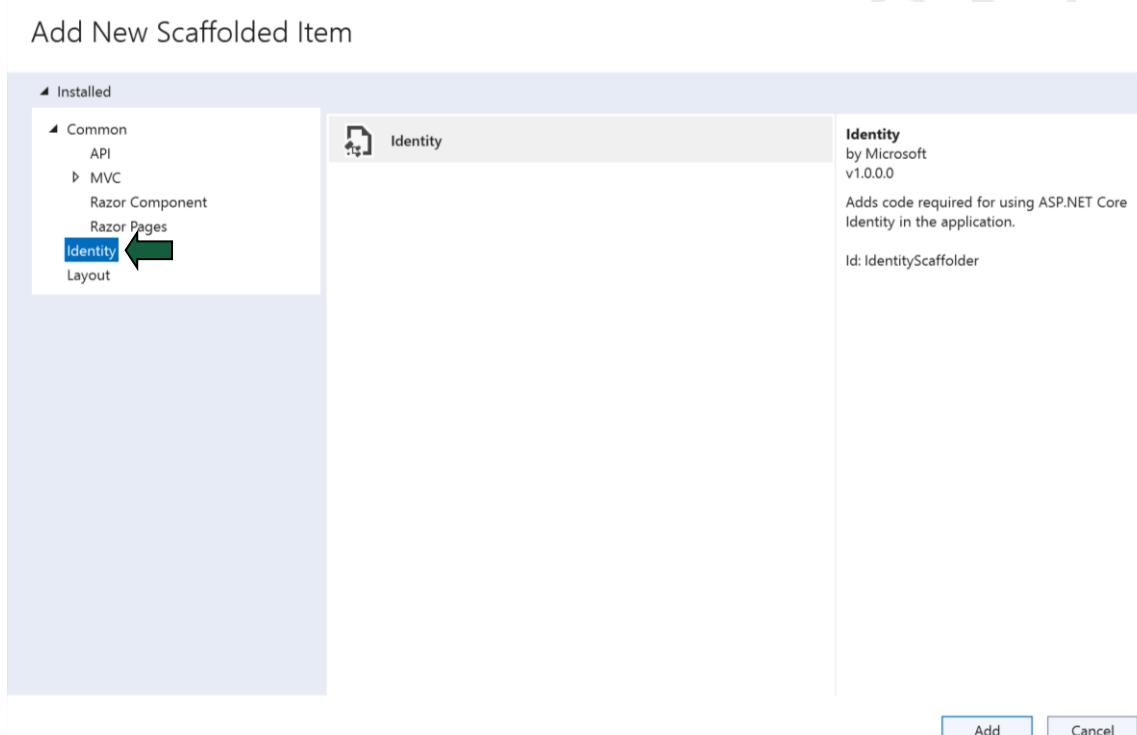
Se ruleaza pe rand:

1. Add-Migration DenumireMigratie
2. Update-Database

## PASUL 9:

Pentru modificarea functionalitatilor implementate in framework, se poate utiliza optiunea **Add Scaffolded Item**, prin intermediul careia se poate aduce in folderele de lucru codul sursa.

Click dreapta pe numele proiectului din Solution Explorer → Add → New Scaffolded Item



Pentru exemplul din cadrul cursului, avem nevoie de functionalitatea de inregistrare. In momentul in care un utilizator se inregistreaza in aplicatie, acesta trebuie sa primeasca un rol. In cazul aplicatiei Engine de stiri, utilizatorii inregistrati o sa primeasca rolul User. Utilizatorii inregistrati pot vedea articolele, pot lasa comentarii, pot edita si sterge propriile comentarii. Utilizatorii devin editori numai in momentul in care Adminul schimba rolul.

Pentru adaugarea rolului de User in momentul inregistrarii, trebuie modificata metoda Register.

## Add Identity

Select an existing layout page, or specify a new one:  
 ~/Views/Shared/\_LayoutNou.cshtml  
 (Leave empty if it is set in a Razor \_viewstart file)

...

Override all files

Choose files to override

<input type="checkbox"/> Account>StatusMessage	<input type="checkbox"/> Account\AccessDenied	<input type="checkbox"/> Account\ConfirmEmail
<input type="checkbox"/> Account\ConfirmEmailChange	<input type="checkbox"/> Account\ExternalLogin	<input type="checkbox"/> Account\ForgotPassword
<input type="checkbox"/> Account\ForgotPasswordConfirmation	<input type="checkbox"/> Account\Lockout	<input type="checkbox"/> Account>Login
<input type="checkbox"/> Account\LoginWith2fa	<input type="checkbox"/> Account\LoginWithRecoveryCode	<input type="checkbox"/> Account\Logout
<input type="checkbox"/> Account\Manage\Layout	<input type="checkbox"/> Account\Manage\ManageNav	<input type="checkbox"/> Account\Manage>StatusMessage
<input type="checkbox"/> Account\Manage\ChangePassword	<input type="checkbox"/> Account\Manage>DeletePersonalData	<input type="checkbox"/> Account\Manage\Disable2fa
<input type="checkbox"/> Account\Manage\DownloadPersonalData	<input type="checkbox"/> Account\Manage>Email	<input type="checkbox"/> Account\Manage\EnableAuthenticator
<input type="checkbox"/> Account\Manage\ExternalLogins	<input type="checkbox"/> Account\Manage\GenerateRecoveryCodes	<input type="checkbox"/> Account\Manage\Index
<input type="checkbox"/> Account\Manage\PersonalData	<input type="checkbox"/> Account\Manage\ResetAuthenticator	<input type="checkbox"/> Account\Manage\SetPassword
<input type="checkbox"/> Account\Manage>ShowRecoveryCodes	<input type="checkbox"/> Account\Manage\TwoFactorAuthentication	<input checked="" type="checkbox"/> Account\Register ←
<input type="checkbox"/> Account\RegisterConfirmation	<input type="checkbox"/> Account\ResendEmailConfirmation	<input type="checkbox"/> Account\ResetPassword
<input type="checkbox"/> Account\ResetPasswordConfirmation		

Data context class:   ←

User class:   ←

Codul inclus se poate accesa din Solution Explorer → Areas → Identity → Pages → Account → Register.cshtml → Register.cshtml.cs\

```
// PASUL 9 - useri si roluri (adaugarea rolului la inregistrare)
await _userManager.AddToRoleAsync(user, "User");
```

## PASUL 10:

Ultimul pas este reprezentat de configurarea managerului de useri si roluri. In Controller-ul **ArticlesController** se implementeaza urmatoarea secenta de cod:

```
public class ArticlesController : Controller
{
    private readonly ApplicationDbContext db;
    private readonly UserManager< ApplicationUser> _userManager;
    private readonly RoleManager< IdentityRole > _roleManager;

    public ArticlesController(
        ApplicationDbContext context,
        UserManager< ApplicationUser > userManager,
        RoleManager< IdentityRole > roleManager)
    {
        db = context;
        _userManager = userManager;
        _roleManager = roleManager;
    }

    ...
}
```

Documentatie Microsoft – User Manager: <https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.identity.usermanager-1?view=aspnetcore-6.0>

Documentatie Microsoft – Role Manager: <https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.identity.rolemanager-1?view=aspnetcore-6.0>

## Atributul [Authorize]

Pentru a restrictiona accesul utilizatorilor la metodele din cadrul unui Controller, se utilizeaza atributul **[Authorize]** la nivelul Controller-ului respectiv.

Pentru a restrictiona accesul in cadrul fiecarei metode, se utilizeaza acelasi atribut, astfel:

```
[Authorize(Roles = "User,Editor,Admin")]
public IActionResult Index()
{ ... }
```

Doar utilizatorii cu rolul User, Editor sau Admin, pot accesa metoda Index.

## Implementare New, Edit si Delete utilizand roluri

In cazul metodelor de adaugare, editare si stergere trebuie sa se tina cont si de id-ul utilizatorului care adauga, editeaza sau sterge articolul. In cazul adaugarii o sa se preia id-ul utilizatorului pentru adaugarea lui in baza de date. In cazul editarii se verifica daca utilizatorul care doreste sa editeze articolul este utilizatorul caruia ii apartine articolul respectiv (adica utilizatorul care a postat articolul). Asemanator se procedeaza si in cazul stergerii unui articol. Un user poate sterge un articol doar daca respectivul articol ii apartine. Un utilizator nu poate sterge articole postate in platforma de alti utilizatori.

### Exemplu implementare metoda New cu Post din ArticlesController:

```
[Authorize(Roles = "Editor,Admin")]
[HttpPost]
public IActionResult New(Article article)
{
    article.Date = DateTime.Now;
    article.UserId = _userManager.GetUserId(User);
```

```

    if (ModelState.IsValid)
    {
        db.Articles.Add(article);
        db.SaveChanges();
        TempData["message"] = "Articolul a fost adaugat";
        return RedirectToAction("Index");
    }
    else
    {
        article.Categ = GetAllCategories();
        return View(article);
    }
}

```

In exemplul anterior se poate observa cum cu ajutorul managerului de utilizatori este preluat id-ul utilizatorului curent, folosind metoda `GetUserId`. Astfel, în momentul inserării articolelor în bază de date, se inserează și id-ul utilizatorului care a postat articolul respectiv.

```
_userManager.GetUserId(User);
```

In cazul editării, se verifică dacă utilizatorul care dorește să modifice articolul este utilizatorul care a postat articolul. Preluarea id-ului utilizatorului curent se realizează tot cu ajutorul metodei `GetUserId`. De asemenea, administratorul având drepturi deplină asupra aplicației, se verifică și rolul. Dacă utilizatorul are rolul Admin, atunci el poate edita articolul. Pentru verificarea rolului se utilizează metoda `IsInRole("NumeRol")`

### Edit - HttpGet

```

...
if (article.UserId == _userManager.GetUserId(User) ||
User.IsInRole("Admin"))
{
    return View(article);
}

else
{
    TempData["message"] = "Nu aveti dreptul sa faceti
modificari asupra unui articol care nu va apartine";
    return RedirectToAction("Index");
}
...

```

La fel se procedeaza si in cazul editarii cu `HttpPost`, dar si in cazul stergerii. Se verifica daca utilizatorul care intentioneaza sa editeze sau sa stearga articolul este userul care a creat respectivul articol. In plus, se verifica si rolul utilizatorului care face actiunea. Daca acesta este Admin, atunci poate face C.R.U.D. asupra oricarei entitati din aplicatie.

# Dezvoltarea Aplicatiilor Web utilizand ASP.NET Core MVC

## Laborator 9

---

### EXERCITII:

1. Pornind de la versiunea implementata in laboratorul anterior, implementati sistemul de autentificare impreuna cu asocierea dintre roluri si utilizatori (**VEZI Curs 9 – Secțiunea Sistemul de autentificare**).  
Urmati toti pasii din Cursul 9.
  
2. Se considera urmatoarele cerinte:
  - sa existe cel putin 4 tipuri de utilizatori: vizitator neinregistrat, utilizator inregistrat, editor si administrator;
  - orice utilizator poate vizualiza stirile aparute pe site. Pe pagina principala vor aparea stirile cele mai recente;
  - stirile vor fi impartite pe categorii (create dinamic): stiinta, tehnologie, sport, etc, existand posibilitatea de adaugare a noi categorii (administratorul poate face CRUD pe categorii);
  - editorii se ocupa de publicarea stirilor noi si pot vizualiza, edita, sterge propriile stiri;
  - utilizatorii pot adauga comentarii la stirile aparute, isi pot sterge si edita propriile comentarii;
  - stirile pot fi cautate prin intermediul unui motor de cautare propriu;
  - administratorii se ocupa de buna functionare a intregii aplicatii (ex: pot face CRUD pe stiri, pe categorii, etc.) si pot activa sau revoca drepturile utilizatorilor si editorilor;

## Implementare – modificari asupra entitatii Article

### Index

Metoda Index se ocupa de preluarea tuturor articolelor din baza de date, impreuna cu categoria din care fac parte. Toate rolurile trebuie sa aiba acces la acesta metoda. Se utilizeaza atributul **[Authorize]**.

```
[Authorize(Roles = "User,Editor,Admin")]
```

De asemenea, se utilizeaza acest atribut si la nivel de Controller, astfel incat doar utilizatorii inregistrati sa aiba acces in aplicatie.

In momentul afisarii articolelor, sa se afiseze si utilizatorul care a publicat fiecare articol. Sa se modifice View-ul asociat metodei.

Pentru verificare sa se adauge manual in baza de date toate intrarile necesare.

### Show

Metoda Show se ocupa se afisarea unui singur articol, in functie de id-ul sau, impreuna cu utilizatorul care a postat articolul respectiv, dar si cu categoria din care face parte. Toate rolurile trebuie sa aiba acces la aceasta metoda.

### New

Un articol poate fi adaugat in baza de date doar de utilizatorii cu rolul Editor sau Admin. Metoda New se ocupa de adaugarea unui nou articol in baza de date. Doar editorii si administratorii pot adauga articole in platforma. In momentul adaugarii unui articol (POST), se retine si id-ul userului care a postat articolul.

## Edit

Editorii pot edita propriile stiri, iar utilizatorii cu rolul Admin pot edita toate articolele existente in platforma. In momentul in care un utilizator care nu are dreptul la editare incearca sa editeze un articol, acesta o sa primeasca un mesaj de tipul “Nu aveti dreptul sa faceti modificari asupra unui articol care nu va apartine”.

## Delete

Editorii isi pot sterge propriile articole, iar utilizatorii cu rolul Admin pot sterge orice articol existent in aplicatie. In cazul in care un utilizator rau intentionat incearca sa stearga un articol, acesta o sa primeasca un mesaj corespunzator: “Nu aveti dreptul sa stergeti un articol care nu va apartine”.

3. Sa se modifice codul existent in **Show**, atat in ArticlesController, cat si in View-ul Show asociat, astfel incat butoanele de **editare si stergere** sa fie vizibile doar pentru utilizatorii cu rolul Admin si Editor. Administratorul poate avea butoanele vizibile pentru orice articol, iar Editorul o sa le aiba vizibile doar pentru articolele care ii aparțin. Utilizatorul cu rolul User nu o sa vada aceste butoane.
  
4. Sa se modifice View-urile **Index** si **Show** din **ArticlesController**, astfel incat sa nu mai existe legatura catre paginile de afisare a tuturor articolelor, a tuturor categoriilor sau catre pagina de adaugare a unui nou articol. Vom proceda astfel: aceste legaturi o sa fie adaugate in meniu din Shared -> \_Layout.cshtml.

Utilizatorii cu rolul **User** o sa vada in meniu doar link-ul “Afisare articole”, utilizatorii cu rolul **Editor** o sa vada in meniu link-urile “Afisare articole” si “Adaugare articol”, iar userul cu rolul **Admin** o sa vada in aplicatie “Afisare articole”, “Adaugare articol”, “Afisare categorii”.

Se utilizeaza in pagina **\_Layout.cshtml** urmatoarele verificari, dupa caz:

```
@if (User.IsInRole("Admin")) { ... }
```

```
@if (User.IsInRole("Editor")) { ... }
```

Pentru stilizarea meniului putem utiliza codul implementat in laboratorul anterior:

```
.navbar {
    background-color: #dfe6e9 !important;
    border: none !important;
    box-shadow: 0 3px 12px #636e72 !important;
}

.a.nav-link {
    color: #2c2c2c !important;
}

.a.nav-link:hover {
    color: #00b894 !important;
}

.logoutbtn {
    color: #2c2c2c !important;
}

.logoutbtn:hover {
    color: #00b894 !important;
}
```

## Implementare – modificari asupra entitatii Category

Utilizatorii cu rolul **Admin** sunt cei care pot face C.R.U.D. pe categorii.

- stirile vor fi impartite pe categorii (create dinamic): stiinta, tehnologie, sport, etc, existand posibilitatea de adaugare a noi categorii (administratorul poate face CRUD pe categorii);
- administratorii se ocupă de buna funcționare a întregii aplicații (ex: pot face CRUD pe stiri, pe categorii, etc.)

Modificati CategoriesController, astfel incat sa implementati functionalitatea de mai sus.

## Implementare – modificari asupra entitatii Comment

Toti utilizatorii autentificati pot lasa comentarii, indiferent de rol. Fiecare utilizator poate edita si sterge doar comentariile proprii, iar administratorul poate edita si sterge orice comentariu.

### **! OBSERVATIE**

Clasa ApplicationUser poate contine atribute suplimentare fata de cele oferite de framework. De asemenea, trebuie definite si proprietatile specifice:

```
public class ApplicationUser : IdentityUser
{
    public virtual ICollection<Comment>? Comments { get; set; }

    public virtual ICollection<Article>? Articles { get; set; }
}
```

# Dezvoltarea Aplicatiilor Web utilizand ASP.NET Core MVC

## Curs 10

---

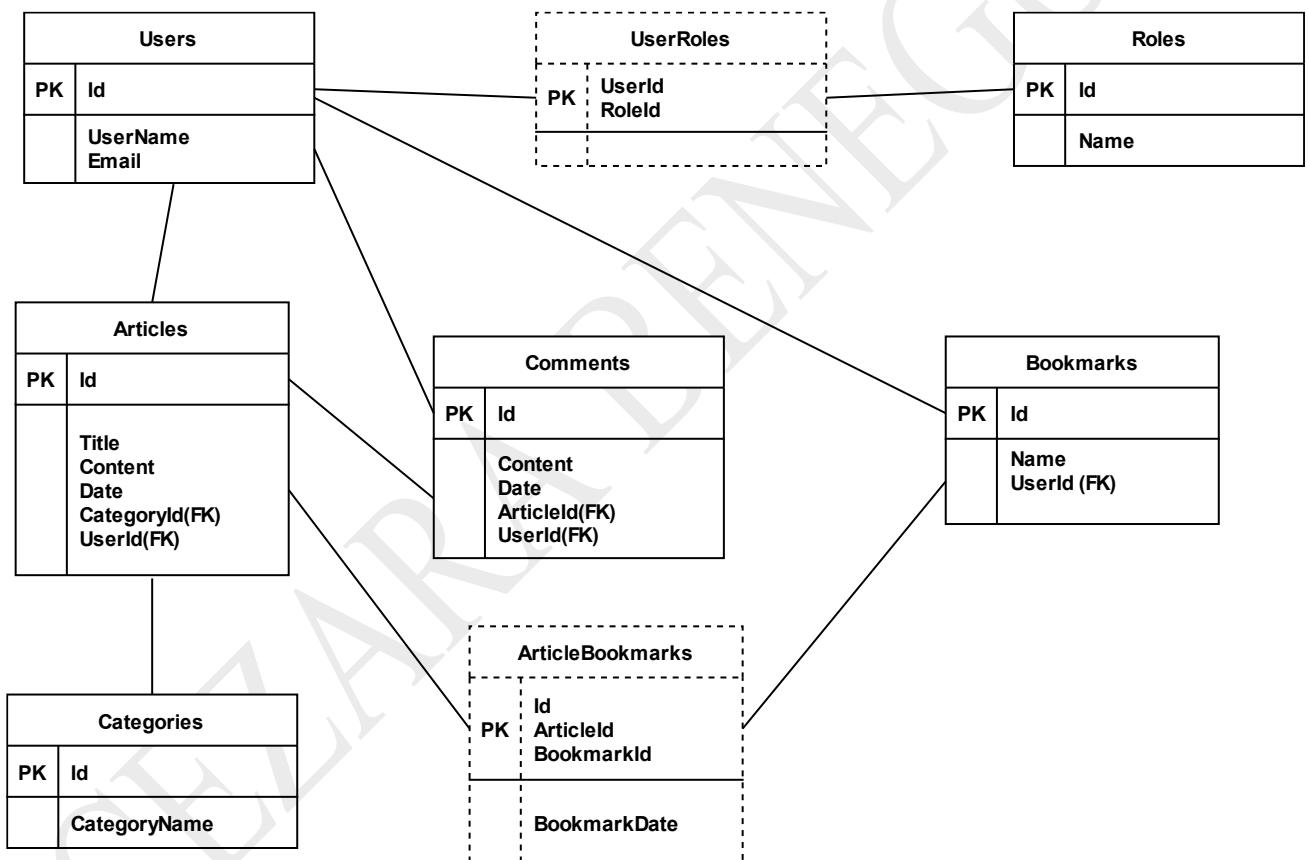
### Cuprins

Implementarea relatiei many-to-many .....	2
Descrierea diagramei.....	2
Implementarea claselor .....	4
Stocarea fisierelor in baza de date .....	8

# Implementarea relatiei many-to-many

## Descrierea diagramei

Pentru implementarea si exemplificarea relatiei many-to-many vom utiliza urmatoarea diagrama conceptuala. Se va extinde diagrama proiectata in cursurile anterioare, prin adaugarea unui nou tabel → Bookmark



Relatia many-to-many se afla intre Article si Bookmark, avand urmatoarele specificatii:

- Un utilizator isi poate crea propriile colectii (Bookmarks)

- În momentul în care un utilizator își adaugă colectii, acesta o să aibă acces doar la colectiile pe care le-a creat, existând posibilitatea editării și a stergerii lor
- Atât utilizatorii cu rolurile User sau Editor, cât și utilizatorii cu rolul Admin, pot crea propriile colectii. Utilizatorii cu rolul User sau Editor o să aibă acces doar la colectiile create de ei, iar utilizatorii cu rolul Admin o să aibă acces la toate colectiile existente în platformă
- Dupa crearea colectiilor, utilizatorii își pot adăuga articole în colectii. Articolele pe care le pot adăuga trebuie să existe în platformă. El doar selectează un articol și îl adăuga într-o colecție. Utilizatorii nu pot vedea colectiile altor utilizatori. Fiecare utilizator are acces doar la colectiile sale
- Un articol poate face parte din mai multe colectii, iar o colecție poate contine mai multe articole

Se consideră urmatoarele clase: **Bookmark**, **ArticleBookmark** (**tabelul asociativ**) cu urmatoarele proprietăți:

#### **Bookmark:**

- **Id** – int → id-ul colecției (cheie primară)
- **Name** – string → denumirea colecției, fiind o proprietate obligatorie (Required)
- **UserId** – string → cheie externă – reprezintă utilizatorul care a creat colecția

#### **ArticleBookmark:**

- **Id, ArticleId, BookmarkId** – int → cheia primară compusă
- **Id** – valoare unică, având auto-increment
- **BookmarkDate** – DateTime → Data și ora la care a fost adăugat un articol în cadrul unei colectii

## Implementarea claselor

Pentru implementarea claselor se procedeaza astfel:

### **Bookmark.cs**

```
public class Bookmark
{
    [Key]
    public int Id { get; set; }

    [Required(ErrorMessage = "Numele colectiei este obligatoriu")]
    public string Name { get; set; }

    public string? UserId { get; set; }

    public virtual ApplicationUser? User { get; set; }

    public virtual ICollection<ArticleBookmark>? ArticleBookmarks
    { get; set; }
}
```

### **Article.cs**

```
public class Article
{
    ...
    public virtual ICollection<ArticleBookmark>? ArticleBookmarks
    { get; set; }
}
```

## ArticleBookmark.cs

```
public class ArticleBookmark
{
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }

    public int? ArticleId { get; set; }

    public int? BookmarkId { get; set; }

    public virtual Article? Article { get; set; }

    public virtual Bookmark? Bookmark { get; set; }

    public DateTime BookmarkDate { get; set; }
}
```

## ApplicationUser.cs

```
public class ApplicationUser : IdentityUser
{
    public virtual ICollection<Comment>? Comments { get; set; }

    public virtual ICollection<Article>? Articles { get; set; }

    public virtual ICollection<Bookmark>? Bookmarks { get; set; }
}
```

## ApplicationDbContext.cs

```
public class ApplicationDbContext :  
IdentityDbContext<ApplicationUser>
{
    public  
ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)  
: base(options)
{}
```

```

public DbSet<ApplicationUser> ApplicationUsers { get; set; }

public DbSet<Article> Articles { get; set; }

public DbSet<Category> Categories { get; set; }

public DbSet<Comment> Comments { get; set; }

public DbSet<Bookmark> Bookmarks { get; set; }

public DbSet<ArticleBookmark> ArticleBookmarks { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    // definire primary key compus
    modelBuilder.Entity<ArticleBookmark>()
        .HasKey(ab => new { ab.Id, ab.ArticleId, ab.BookmarkId });

    // definire relatii cu modelele Bookmark si Article (FK)
    modelBuilder.Entity<ArticleBookmark>()
        .HasOne(ab => ab.Article)
        .WithMany (ab => ab.ArticleBookmarks)
        .HasForeignKey(ab => ab.ArticleId);

    modelBuilder.Entity<ArticleBookmark>()
        .HasOne(ab => ab.Bookmark)
        .WithMany(ab => ab.ArticleBookmarks)
        .HasForeignKey(ab => ab.BookmarkId);
}
}

```

Se pot adauga si proprietati suplimentare in clasa ApplicationUser, extinzand astfel clasa. In exemplul urmator se adauga doua attribute → **FirstName** si **LastName**.

### ApplicationUser.cs

```
public class ApplicationUser : IdentityUser
{
    public virtual ICollection<Comment>? Comments { get; set; }

    public virtual ICollection<Article>? Articles { get; set; }

    public virtual ICollection<Bookmark>? Bookmarks { get; set; }

    public string? FirstName { get; set; }

    public string? LastName { get; set; }

    [NotMapped]
    public IEnumerable<SelectListItem>? AllRoles { get; set; }

}
```

### ! OBSERVATIE

Dupa modificarea claselor se executa migratiile.

*Add-Migration NumeMigratie  
Update-Database*

## Stocarea fisierelor in baza de date

```
public class Article
{
    [Key]
    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public DateTime Date { get; set; }

    // Adaugam un string unde vom salva calea imaginii pentru articol
    public string Image { get; set; }
}
```

### View-ul asociat:

```
<form enctype="multipart/form-data" asp-controller="Articles" asp-action="UploadImage">

    <input class="form-control" placeholder="Article Name" type="text" name="Title" />

    <br />

    <textarea class="form-control" placeholder="Article Content" style="width: 100%; height: 100px;" name="Content"></textarea>

    <br />

    <input class="form-control" placeholder="Date" type="text" name="Date" />

    <br />

    <input class="form-control" type="file" name="ArticleImage" />

    <br />

    <input class="btn btn-success" type="submit" value="Upload" />

</form>
```

## Controller:

```
// Variabila locala de tip AppDbContext

private AppDbContext _context;
private IWebHostEnvironment _env;

// In constructor, se face dependency injection
public ArticlesController(AppDbContext context, IWebHostEnvironment env)
{
    // Alocam conexiunea injectata la baza de date unei proprietati
    // locale pentru a fi folosita in metodele controller-ului
    _context = context;
    _env = env;
}

// Afisam view-ul cu form-ul
public IActionResult UploadImage()
{
    return View();
}

// Facem upload la fisier si salvam modelul in baza de date
[HttpPost]

public async Task<IActionResult> UploadImage(Article article,
IFormFile ArticleImage)

{
    // Verificam daca exista imaginea in request (daca a fost
    // incarcata o imagine)

    if (ArticleImage.Length > 0)

    {
        // Generam calea de stocare a fisierului
        var storagePath = Path.Combine(
            _env.WebRootPath, // Luam calea folderului wwwroot
            "images", // Adaugam calea folderului images
            ArticleImage.FileName // Numele fisierului
        );
    }
}
```

```
// General calea de afisare a fisierului care va fi stocata in  
baza de date  
var databaseFileName = "/images/" + ArticleImage.FileName;  
  
// Uploadam fisierul la calea de storage  
using (var fileStream = new FileStream(storagePath,  
FileMode.Create))  
{  
    await ArticleImage.CopyToAsync(fileStream);  
}  
}  
  
// Salvam storagePath-ul in baza de date  
article.Image = databaseFileName;  
_context.Articles.Add(article);  
_context.SaveChanges();  
  
return View();  
}
```

# Dezvoltarea Aplicatiilor Web utilizand ASP.NET Core MVC

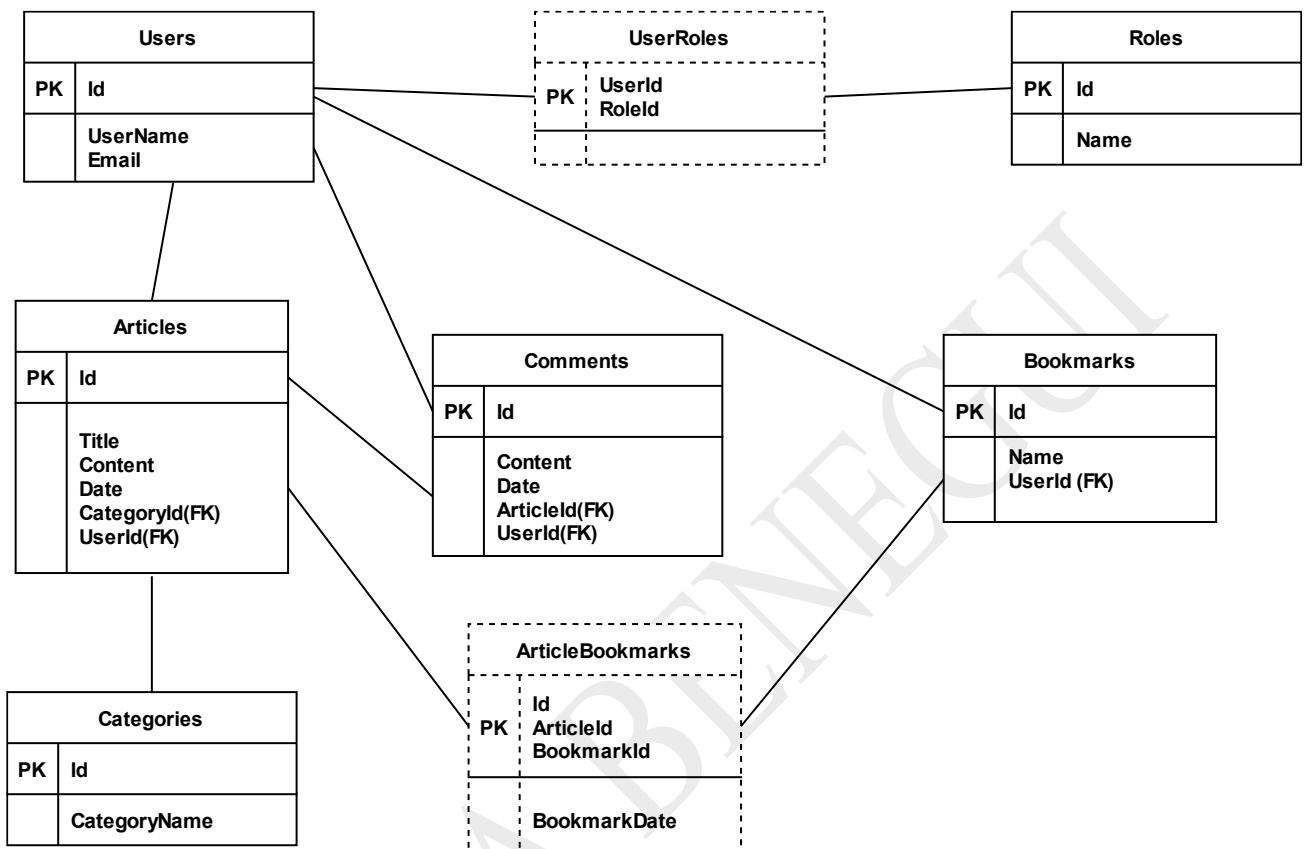
## Laborator 10

---

### EXERCITII:

1. Pornind de la urmatoarele specificatii sa se implementeze posibilitatea de adaugare a articolelor in colectii:
  - Un utilizator isi poate crea propriile colectii (Bookmarks)
  - In momentul in care un utilizator isi adauga colectii, acesta o sa aiba acces doar la colectiile pe care le-a creat, existand posibilitatea editarii si a stergerii lor
  - Atat utilizatorii cu rolurile User sau Editor, cat si utilizatorii cu rolul Admin, pot crea propriile colectii. Utilizatorii cu rolul User sau Editor o sa aiba acces doar la colectiile create de ei, iar utilizatorii cu rolul Admin o sa aiba acces la toate colectiile existente in platforma
  - Dupa crearea colectiilor, utilizatorii isi pot adauga articole in colectii. Articolele pe care le pot adauga trebuie sa existe in platforma. Ei doar selecteaza un articol si il adauga intr-o colectie. Utilizatorii nu pot vedea colectiile altor utilizatori. Fiecare utilizator are acces doar la colectiile sale
  - Un articol poate face parte din mai multe colectii, iar o colectie poate contine mai multe articole

Se considera Diagrama Conceptuala:



Se modifica pe rand clasele Bookmark, Article, ArticleBookmark, ApplicationUser, ApplicationUser si contextul bazei de date ApplicationDbContext.

Dupa modificarile claselor se realizeaza o migratie.

Dupa migratie se vor implementa cerintele anterioare.

2. Sa se implementez functionalitatea prin care utilizatorii cu rolul Admin pot activa sau revoca drepturile celorlalți utilizatori. De asemenea, administratorii pot edita și sterge ceilalți utilizatori din aplicație.

CEZARA BENEGUI

# Dezvoltarea Aplicatiilor Web utilizand ASP.NET Core MVC

## Curs 11

---

### Cuprins

Afisarea paginata.....	2
Exemplu de implementare .....	2
Includerea unui editor de text .....	5
Exemplu de implementare .....	6
Functionalitatea de cautare .....	15
Descrierea si implementarea motorului de cautare.....	15
Design-ul intr-o aplicatie Web.....	19
Reguli de baza in design .....	19
User Experience (UX).....	22
Alegerea culorilor potrivite.....	26

## Afisarea paginata

Afisarea paginata este utila in cazul in care trebuie sa afisam un numar mare de elemente, numar care se modifica constant. De exemplu, trebuie sa afisam foarte multe categorii, articole, produse, etc. In acest caz se utilizeaza afisarea paginata, in cadrul careia dezvoltatorii implementeaza afisarea in functie de o serie de reguli. Regulile trebuie stabilite luand in considerare numarul de elemente pe care dorim sa-l afisam, tinand cont si de faptul ca acest numar o sa isi modifice valoarea in functie de actiunile utilizatorilor in cadrul aplicatiei (de ex: utilizatorii pot adauga/sterge articole sau produse).

## Exemplu de implementare

Exemplul urmator este realizat in cadrul aplicatiei Engine de stiri (aplicatia dezvoltata in laborator).

Pentru afisarea paginata o sa se utilizeze din Bootstrap – Pagination → <https://getbootstrap.com/docs/5.2/components/pagination/>

Se vor afisa 3 articole pe pagina, modificandu-se atat metoda Index din ArticlesController, cat si View-ul Index corespunzator.

### Includerea si modificarea componentei din Bootstrap:

```
<nav aria-label="Page navigation example">
  <ul class="pagination">
    <li class="page-item">
      <a class="page-link" href="#" aria-label="Previous">
        <span aria-hidden="true">&laquo;</span>
      </a>
    </li>
    <li class="page-item"><a class="page-link" href="#">1</a></li>
    <li class="page-item"><a class="page-link" href="#">2</a></li>
    <li class="page-item">
      <a class="page-link" href="#" aria-label="Next">
        <span aria-hidden="true">&raquo;</span>
      </a>
    </li>
  </ul>
</nav>
```

Seventa de cod anterioara o sa se modifice astfel. Modificarile sunt scrise cu negru si bold.

```
@* Afisarea paginata a articolelor *@

<div>
    <nav aria-label="Page navigation example">
        <ul class="pagination">
            <li class="page-item">
                <a class="page-link"
                    href="/Articles/Index?page=1" aria-label="Previous">
                    <span aria-hidden="true">&laquo;</span>
                </a>
            </li>

            @for (int i = 1; i <= ViewBag.lastPage; i++)
            {
                <li class="page-item"> <a class="page-link"
                    href="/Articles/Index?page=@i">@(i)</a> </li>
            }

            <li class="page-item">
                <a class="page-link"
                    href="/Articles/Index?page=@(ViewBag.lastPage)" aria-
                    label="Next">
                    <span aria-hidden="true">&raquo;</span>
                </a>
            </li>
        </ul>
    </nav>
</div>
```

## ArticlesController → metoda Index

```
[Authorize(Roles = "User,Editor,Admin")]
public IActionResult Index()
{
    // Alegem sa afisam 3 articole pe pagina
    int _perPage = 3;

    var articles = db.Articles.Include("Category")
        .Include("User").OrderBy(a => a.Date);

    if (TempData.ContainsKey("message"))
    {
        ViewBag.message =
            TempData["message"].ToString();
    }

    // Fiind un numar variabil de articole, verificam de
    fiecare data utilizand
    // metoda Count()

    int totalItems = articles.Count();

    // Se preia pagina curenta din View-ul asociat
    // Numarul paginii este valoarea parametrului page
    din ruta
    // /Articles/Index?page=valoare

    var currentPage =
        Convert.ToInt32(HttpContext.Request.Query["page"]);

    // Pentru prima pagina offsetul o sa fie zero
    // Pentru pagina 2 o sa fie 3
    // Asadar offsetul este egal cu numarul de articole
    care au fost deja afisate pe paginile anterioare
    var offset = 0;

    // Se calculeaza offsetul in functie de numarul
    paginii la care suntem
    if (!currentPage.Equals(0))
    {
        offset = (currentPage - 1) * _perPage;
    }
}
```

```

        // Se preiau articolele corespunzatoare pentru
fiecare pagina la care ne aflam
        // in functie de offset
        var paginatedArticles =
articles.Skip(offset).Take(_perPage);

        // Preluam numarul ultimei pagini

ViewBag.lastPage = Math.Ceiling((float)totalItems /
(float)_perPage);

        // Trimitem articolele cu ajutorul unui ViewBag
catre View-ul corespunzator
ViewBag.Articles = paginatedArticles;

        return View();
}

```

## Includerea unui editor de text

In aceasta sectiune vom integra o componenta externa (componenta 3<sup>rd</sup> party). Din categoria componentelor 3<sup>rd</sup> party fac parte: servere web (Apache, Nginx, IIS, etc), framework-uri (.NET, Rails, Spring, etc), librarii.

Componenta pe care o vom integra este un editor de text open-source, bazat pe Bootstrap si jQuery, numita **Summernote**. Este o librarie de JavaScript folosita pentru a implementa un editor de tipul **WYSIWUG - What You See Is What You Get**

Acest tip de editor ofera posibilitatea prelucrarii textului de catre utilizatorii finali, asemănător cu scrierea textului în Word sau folosind limbaj de markup (HTML).

Pagina oficiala → <https://summernote.org/>

## Exemplu de implementare

Exemplul urmator este realizat in cadrul aplicatiei Engine de stiri (aplicatia dezvoltata in laborator).

Se doreste utilizarea editorului de text atat in momentul adaugarii unui articol, cat si in momentul editarii acestuia.

Se integreaza Summernote accesand link-ul urmator si urmand pasii:

### PASUL 1:

<https://summernote.org/getting-started/#without-bootstrap-lite>

### PASUL 2:

In **\_Layout.cshtml** se include in Head componenta de css:

```
<link href="https://cdn.jsdelivr.net/npm/summernote@0.8.18/dist/summernote-lite.min.css" rel="stylesheet">
```

```

2   <html lang="en">
3     <head>
4       <meta charset="utf-8" />
5       <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6       <title>@ ViewData["Title"] - ArticlesApp</title>
7       <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
8       <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.10.1/font/bootstrap-icons.css" />
9
10      <link rel="stylesheet" href="~/ArticlesApp.styles.css" asp-append-version="true" />
11      <link rel="stylesheet" href="~/css/site.css" asp-append-version="true" />
12
13
14      <!-- se include summernote css -->
15      <link href="https://cdn.jsdelivr.net/npm/summernote@0.8.18/dist/summernote-lite.min.css" rel="stylesheet">
16
17    </head>
18    <body>
19      <header>
20        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow" 21 <div class="container-fluid">
22          <a class="navbar-brand" asp-area="" asp-controller="Articles" asp-action="Index">ArticlesApp</a>

```

### PASUL 3:

In **\_Layout.cshtml** se include in Body componenta de js:

```
<script src="https://cdn.jsdelivr.net/npm/summernote@0.8.18/dist/summernote-lite.min.js"></script>
```



```

94     <footer class="border-top footer text-muted">
95         <div class="container">
96             &copy; 2022 - ArticlesApp - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
97         </div>
98     </footer>
99
100    <script src="~/lib/jquery/dist/jquery.min.js"></script>
101    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
102
103    <script src="https://cdn.jsdelivr.net/npm/summernote@0.8.18/dist/summernote-lite.min.js"></script> ←
104
105    <script src="~/js/site.js" asp-append-version="true"></script>
106
107    @await RenderSectionAsync("Scripts", required: false)
108
109
110
111    </body>
112    </html>

```

### PASUL 4:

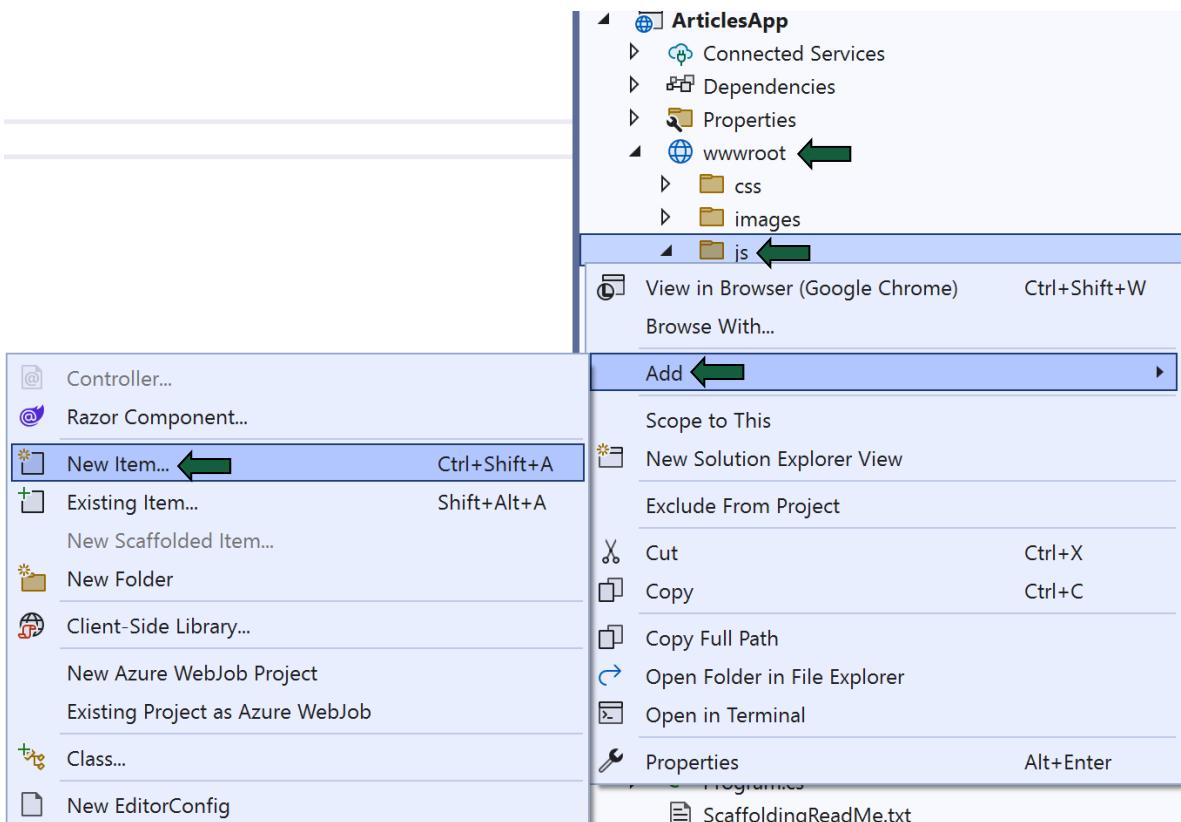
Editorul se initializeaza adaugand urmatoarea secventa de cod intr-un fisier de tip JavaScript (js).

```

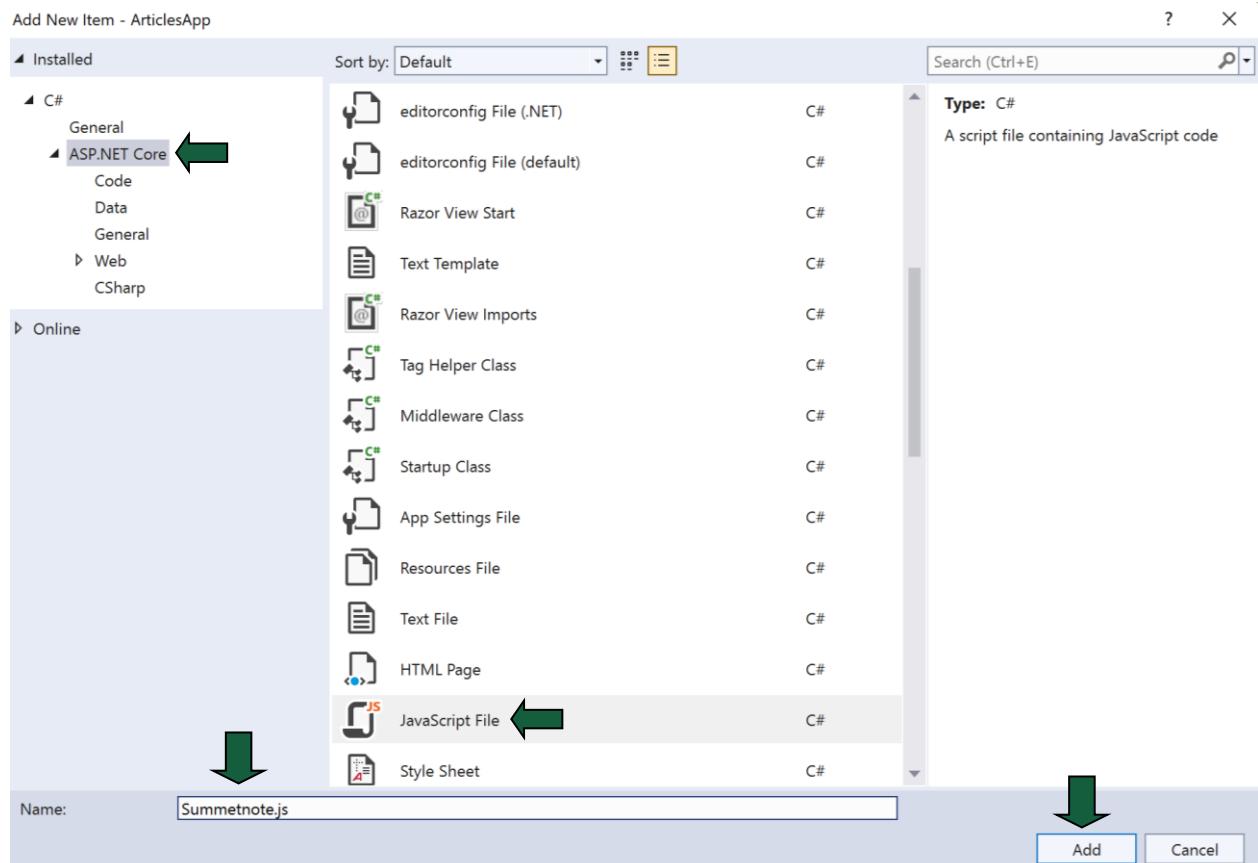
$(document).ready(function () {
    $('.summernote').summernote({
        height: 300,
        minHeight: 200,
        focus: true,
    });
});

```

## PASUL 5:



## PASUL 6:



## PASUL 7:

Scriptul-ul se include in \_Layout in Body:

```

93
94     <footer class="border-top footer text-muted">
95         <div class="container">
96             &copy; 2022 - ArticlesApp - <a asp-area="" asp-controller="Home" asp-action="Privacy">P
97             </a>
98         </footer>
99
100
101
102
103
104
105
106     <script src="~/lib/jquery/dist/jquery.min.js"></script>
107     <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
108
109     <script src="https://cdn.jsdelivr.net/npm/summernote@0.8.18/dist/summernote-lite.min.js"><scri
110
111     <script src="~/js/site.js" asp-append-version="true"></script>
112     <script src="~/js/Summernote.js" asp-append-version="true"></script> ←
113
114     @await RenderSectionAsync("Scripts", required: false)
115
116
117     </body>
118
119     </html>

```

## PASUL 8:

Pentru utilizarea editorului se foloseste clasa definita in fisierul Summernote.js

```
@Html.TextArea("Content", null, new { @class = "summernote" })
```



Dupa integrarea editorului se observa o problema de afisare:

# Adaugare articol

Titlu Articol

Continut Articol

The screenshot shows a rich text editor window. At the top is a toolbar with icons for bold (B), italic (I), underline (U), font color (color palette), font family (sans-serif dropdown), font size (A dropdown), and alignment (left, center, right, justify). Below the toolbar is a content area with a cursor. Three green arrows point upwards from the bottom of the page towards the toolbar, specifically highlighting the font family, font size, and alignment dropdowns.

Pentru afisarea corecta a dropdown-ului se utilizeaza (astfel incat sa nu afiseze acele sageti dupicat):

```
.note-editor .dropdown-toggle::after {  
    display: none;  
}
```

## PASUL 9:

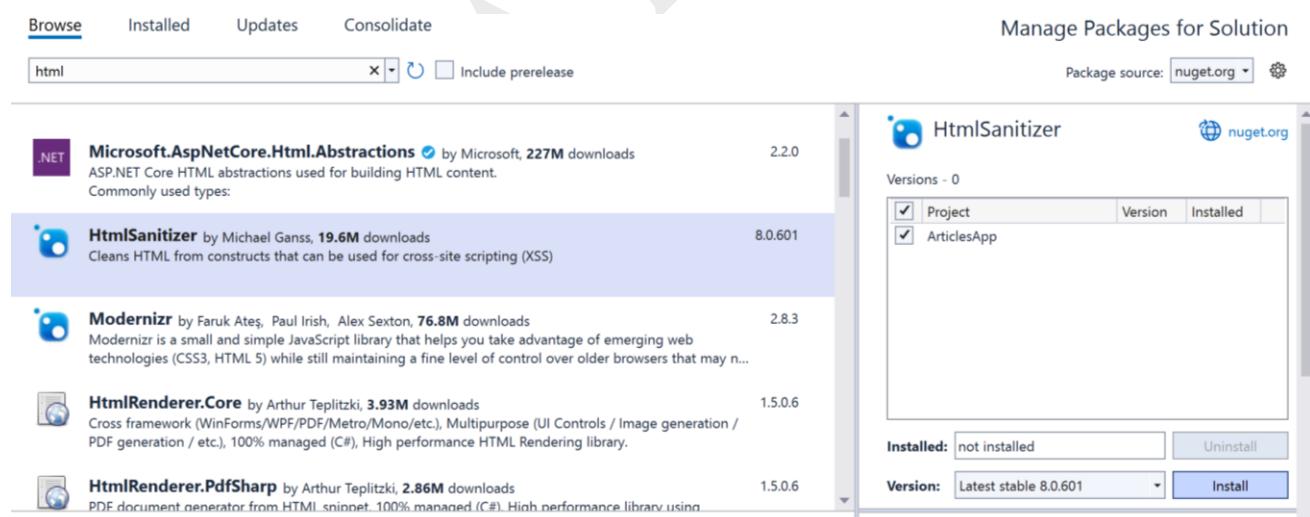
Pentru afisare vom utiliza `@Html.Raw()`

```
<div class="card-text">@Html.Raw(Model.Content)</div>
```

## PASUL 10:

Prevenire XSS (Cross Site Scripting). XSS presupune posibilitatea inserarii unui cod, de catre o persoana neautorizata, intr-o pagina web. In cazul exemplului din cadrul cursului, un utilizator neautorizat poate introduce cod JavaScript, in formular, in momentul inserarii unui nou articol in platforma.

Pentru a preveni XSS este nevoie de instalarea unui nou pachet din **NuGet package → HtmlSanitizer**



In cadrul adaugarii si editarii din Controller o sa avem urmatoarele modificari:

### **ArticlesController → metoda New cu POST**

[HttpPost]

```
public IActionResult New(Article article)
{
    var sanitizer = new HtmlSanitizer(); ←
    article.Date = DateTime.Now;
    article.UserId = _userManager.GetUserId(User);

    if (ModelState.IsValid)
    {
        article.Content = sanitizer.Sanitize(article.Content); ←
        article.Content = (article.Content);

        db.Articles.Add(article);
        db.SaveChanges();
        TempData["message"] = "Articolul a fost adaugat";
        return RedirectToAction("Index");
    }
    else
    {
        article.Categ = GetAllCategories();
        return View(article);
    }
}
```

## ArticlesController → metoda Edit cu POST

```
[HttpPost]
[Authorize(Roles = "Editor,Admin")]
public IActionResult Edit(int id, Article requestArticle)
{
    Var sanitizer = new HtmlSanitizer(); ←
    Article article = db.Articles.Find(id);

    if (ModelState.IsValid)
    {
        if (article.UserId == _userManager.GetUserId(User) ||
User.IsInRole("Admin"))
        {
            article.Title = requestArticle.Title;

            requestArticle.Content =
sanitizer.Sanitize(requestArticle.Content); ←

            article.Content = requestArticle.Content;

            article.CategoryId =
requestArticle.CategoryId;
            TempData["message"] = "Articolul a fost
modificat";
            db.SaveChanges();
            return RedirectToAction("Index");
        }
        else
        {
            TempData["message"] = "Nu aveti dreptul sa
faceti modificari asupra unui articol care nu va apartine";
            return RedirectToAction("Index");
        }
    }
    else
    {
        requestArticle.Categ = GetAllCategories();
        return View(requestArticle);
    }
}
```

In cazul editarii, pentru preluarea corecta a continutului articolului din baza de date, trebuie sa utilizam TextArea. Astfel, se inlocuieste helperul @Html.Editor cu helperul @Html.TextArea.

```
@* @Html.Editor("Content", new { htmlAttributes = new { @class = "form-control summernote" } }) *@

@Html.TextArea("Content", Model.Content, new { @class = "form-control summernote" })
```

## Functionalitatea de cautare

Functionalitatea de cautare se afla integrata in orice aplicatie, indiferent de natura ei (aplicatie web, aplicatie mobile, desktop, etc).

Cel mai mare motor de cautare este **GOOGLE**, urmat de Bing si Baidu. Acestea ocupa in momentul de fata primele 3 locuri.

## Descrierea si implementarea motorului de cautare

Pentru a exemplifica integrarea motorului de cautare, o sa utilizam aplicatia dezvoltata in laborator – Engine de stiri. In cadrul aplicatiei se integreaza functionalitatea de cautare a articolelor dupa **titlu**, **continut**, dar si dupa **continutul comentariilor** postate de utilizatori in cadrul articolelor respective.

Se utilizeaza componenta Bootstrap → Input group

<https://getbootstrap.com/docs/5.0/forms/input-group/>

## Implementare:

### View-ul Index

```
<form method="GET">

    <div class="input-group mb-3">

        <input type="text" class="form-control"
placeholder="Search topics or keywords" name="search"
value="@ViewBag.SearchString">

        <button class="btn btn-outline-success"
type="submit">Search</button>

    </div>

</form>
```

### ArticlesController – Metoda Index

```
[Authorize(Roles = "User,Editor,Admin")]
public IActionResult Index()
{
    var articles = db.Articles.Include("Category")
                            .Include("User").OrderBy(a => a.Date);

    var search = "";

    // MOTOR DE CAUTARE

    if (Convert.ToString(HttpContext.Request.Query["search"]) != null)
    {
        // eliminam spatiile libere
        search =
Convert.ToString(HttpContext.Request.Query["search"]).Trim();
```

```

// Cautare in articol (Title si Content)

List<int> articleIds = db.Articles.Where
(
    at => at.Title.Contains(search)
    || at.Content.Contains(search)
).Select(a => a.Id).ToList();

// Cautare in comentarii (Content)
List<int> articleIdsOfCommentsWithSearchString =
db.Comments.Where
(
    c => c.Content.Contains(search)
).Select(c => (int)c.ArticleId).ToList();

// Se formeaza o singura lista formata din toate id-urile
selectate anterior
List<int> mergedIds =
articleIds.Union(articleIdsOfCommentsWithSearchString).ToList();

// Lista articolelor care contin cuvantul cautat
// fie in articol -> Title si Content
// fie in comentarii -> Content
articles = db.Articles.Where(article =>
mergedIds.Contains(article.Id))
    .Include("Category")
    .Include("User")
    .OrderBy(a => a.Date);

}

ViewBag.SearchString = search;

// AFISARE PAGINATA

{ ... implementarea se afla in sectiunea anterioara }

if(search != "")
{
    ViewBag.PaginationBaseUrl = "/Articles/Index/?search="
+ search + "&page";
} else
{
    ViewBag.PaginationBaseUrl = "/Articles/Index/?page";
}

return View();
}

```

Dupa cum se observa in implementarea din cadrul metodei Index din ArticlesController, trebuie sa ne asiguram ca afisarea paginata din cadrul sectiunii anterioare functioneaza si in momentul integrarii motorului de cautare.

Astfel, in cazul in care se foloseste motorul de cautare, URL-ul o sa fie de forma: `/Articles/Index/?search&page` ceea ce inseamna ca trebuie sa avem in ruta atat stringul cautat, cat si numarul paginii.

Afisarea paginata din cadrul View-ului Index o sa se modifice astfel:

```
@* Afisarea paginata a articolelor *@

<div>
    <nav aria-label="Page navigation example">
        <ul class="pagination">
            <li class="page-item">
                <a class="page-link"
                    href="@ViewBag.PaginationBaseUrl=1" aria-label="Previous">
                    <span aria-hidden="true">&laquo;</span>
                </a>
            </li>

            @for (int i = 1; i <= ViewBag.lastPage; i++)
            {
                <li class="page-item"> <a class="page-link"
                    href="@ViewBag.PaginationBaseUrl=@(i)">@(i)</a> </li>
            }

            <li class="page-item">
                <a class="page-link"
                    href="@ViewBag.PaginationBaseUrl=@(ViewBag.lastPage)" aria-
                    label="Next">
                    <span aria-hidden="true">&raquo;</span>
                </a>
            </li>
        </ul>
    </nav>
</div>
```

# Design-ul intr-o aplicatie Web

## Reguli de baza in design

Pentru realizarea aplicatiilor Web, in ceea ce priveste design-ul acestora, se pot respecta anumite reguli:

- **Simplitatea** – prea multa informatie intr-o pagina distrage atentia utilizatorului deoarece acesta trebuie sa citeasca si sa parcurga mult prea multa informatie pana la informatia de care are nevoie. Pastrand paginile simple, aplicatia va fi mult mai usor de folosit.
- **Design-ul este esential** – prima impresie conteaza. Aceasta regula se aplica si in dezvoltarea unei aplicatii web deoarece este primul lucru pe care il observa un utilizator atunci cand acceseaza aplicatia.
- **Culorile sunt foarte importante** – atunci cand alegeti culorile folositi o paleta consistenta de culori. De asemenea, sa va asigurati ca nu exista nuante foarte apropiate care nu pot fi deosebite si mai ales ca exista un contrast puternic intre text si fundal.
- **Navigarea ar trebui sa fie intuitiva** – un utilizator nu trebuie sa caute ceea ce doreste sa acceseze. Paginile trebuie sa fie bine organizate cu un design de tipul top-down, utilizatorii navigand usor printre diferitele sectiuni existente intr-o pagina.
- **Consistenta este extrem de importanta** – utilizatorii nu ar trebui sa aiba sentimentul ca viziteaza un alt site sau o alta aplicatie web de fiecare data cand acceseaza o alta pagina a aplicatiei. Consistenta face navigarea in aplicatie mult mai simpla.
- **Aplicatia trebuie sa fie responsive** – utilizatorii acceseaza aplicatia utilizand o varietate de device-uri, de la smartphone-uri la calculatoare personale, de aceea este esential ca aplicatia sa se incadreze oricarei

rezolutii. CSS media queries sunt ideale pentru realizarea rapida a unei aplicatii web responsive.

- **Utilizarea unui continut real in momentul dezvoltarii design-ului.**  
Orice aplicatie este bazata pe continut si se dezvolta in jurul continutului. De cele mai multe ori atunci cand se dezvolta design-ul nu se acorda importanta continutului, folosindu-se Lorem Ipsum in locul textului real si placeholder in locul imaginilor. Chiar daca totul arata bine in timpul dezvoltarii design-ului, nu o sa mai fie la fel atunci cand aplicatia va contine datele reale. Scopul in dezvoltarea unei aplicatii web este de a fi cat mai aproape de experienta reala a utilizatorului
- **Fontul** – in general fonturile Sans Serif, cum ar fi Arial si Verdana, sunt mai usor de citit intr-o aplicatie web, fiind fonturi fara finisaje decorative. Dimensiunea fontului pentru citirea cu usurinta este de 16px

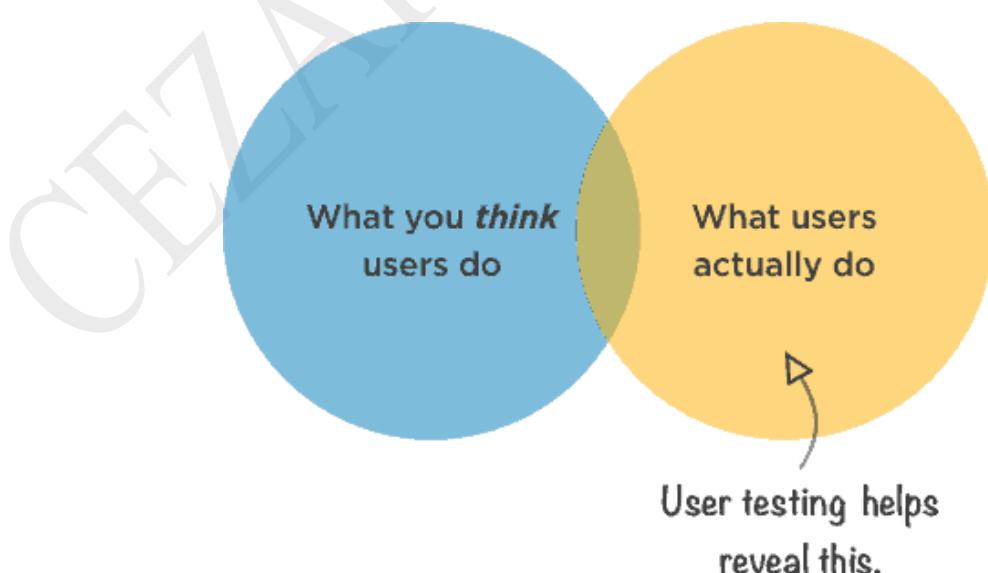


- **Imaginiile** – ajuta la o buna interactiune cu utilizatorul
- **“F” Pattern Design** – informatiile sa fie prezentate, in ordinea relevantei, de la stanga la dreapta si de sus in jos, un studiu aratand ca partea de sus-stanga a ecranului este mult mai vizualizata decat cea de jos-dreapta

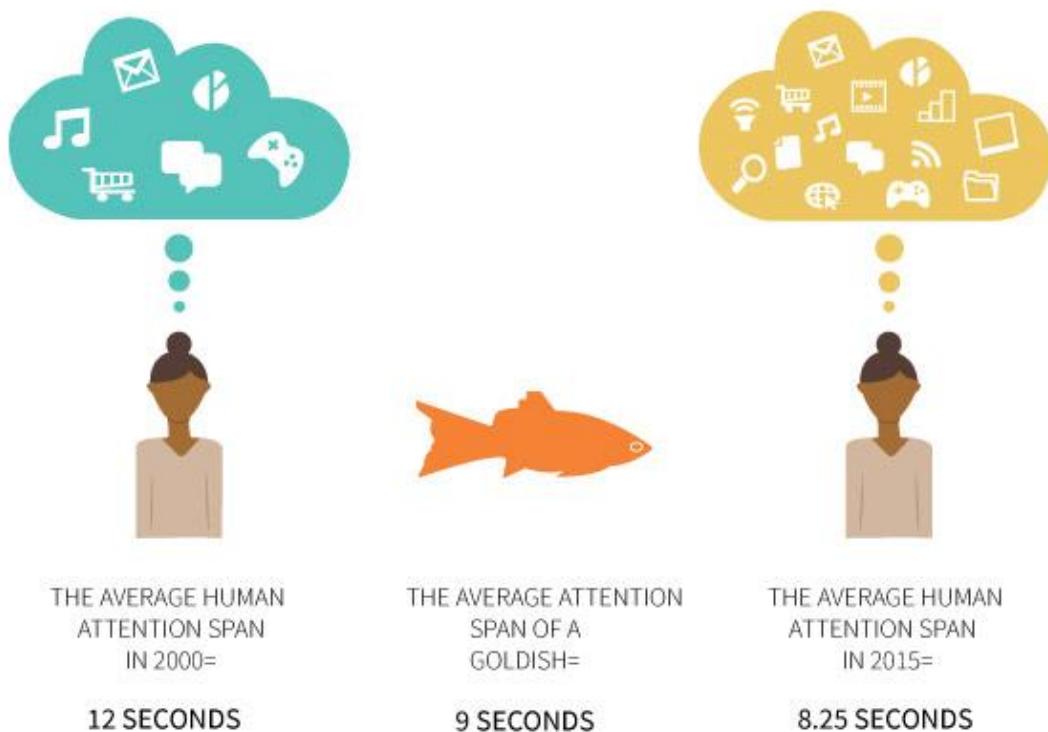
## User Experience (UX)

In primul rand, interfata cu utilizatorul (User Interface) este o parte din UX, existand o diferență semnificativă între cele două. Interfața cu utilizatorul este spațiul unde utilizatorul interacționează cu aplicația, cu diferite componente ale aplicației, iar User Experience este rezultatul final pe care îl produce interacțiunea cu aplicația.

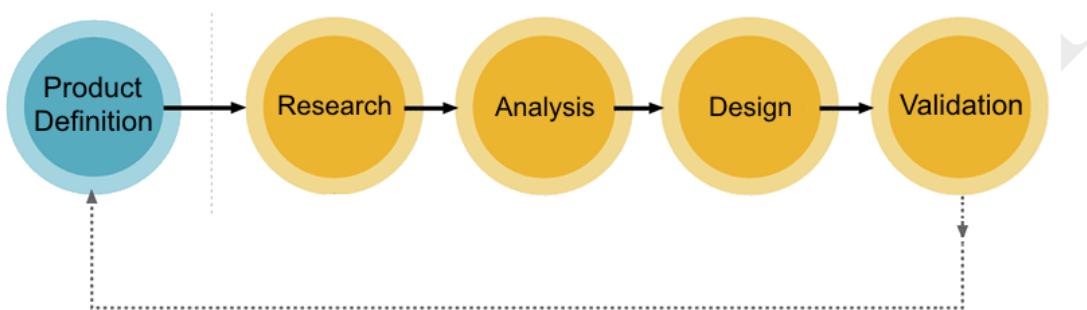
- **Scopul aplicației** – cel mai important lucru este scopul aplicației; Cine o să fie utilizatorii finali? Care sunt nevoile lor? Ce își doresc ei? De aceea, dezvoltarea unei aplicații poate începe cu un “user research”, fiind un proces esențial pentru UX.
- **Tu nu ești utilizatorul final** – este un lucru esențial în dezvoltarea unei aplicații. Dezvoltatorul nu este utilizatorul final, iar cel mai important lucru este testarea de către un utilizator real. Un dezvoltator nu se va comporta niciodată ca un utilizator real deoarece are experiența și va utiliza aplicația exact cum a fost dezvoltată. Utilizatorii care vor folosi aplicația au moduri diferite de gândire, un background diferit, experiență diferită sau chiar scopuri diferite, ceea ce face testarea cu utilizatori reali cea mai bună formă de testare pentru o aplicație web (usability testing)



- **Adaptarea design-ului pentru atentia de scurta durata** – nu incarcati utilizatorii cu foarte multa informatie. O perioada de atentie inseamna acea perioada de timp in care cineva se concentreaza asupra unei sarcini fara sa fie distras. Un studiu realizat de Microsoft in 2015 a aratat faptul ca media atentiei umane a scazut de la 12 secunde la 8 secunde, ceea ce inseamna ca avem o perioada de atentie mai scurta decat cea a unui pestisor auriu. Acest lucru inseamna ca este necesar ca designerii sa afiseze oamenilor informatiile necesara cat mai repede posibil. De aceea, lucrurile care nu sunt necesare pot fi eliminate. Acest lucru nu inseamna ca trebuie sa limitam experientele utilizatorilor intr-o aplicatie web, ci doar ca informatiile trebuie sa fie relevante.



- **UX depinde de fiecare proiect in parte** – nu exista un UX general care poate fi aplicat oricarui proiect. Fiecare proiect este unic, are cerinte unice, ceea ce duce la un UX diferit fiecarei aplicatii. De exemplu, atunci cand dezvoltati un nou proiect este necesara alocarea unui timp suplimentar pentru a cerceta ce tipuri de utilizatori vor accesa aplicatia, dar si care sunt specificatiile aplicatiei.



- **Prevenirea erorilor este mai usoara decat rezolvarea lor** – multe erori apar din cauza utilizatorilor (email incorrect, parole care nu respecta anumite reguli, etc.). In acest caz aplicatia poate fi dezvoltata oferind sugestii de completare si notificand utilizatorii de fiecare data.

### Register Yourself

jamie+3p@baymard.com !

You have entered an invalid e-mail address. Please try again.

Name	Surname
Birth Date	<input type="button" value=""/>

- **Afisarea feedback-ului** – utilizatorii trebuie sa fie mereu informati de tot ceea ce se intampla in cadrul unei aplicatii: mesaje dupa trimiterea formularelor, mesaje in momentul editarii/stergerii datelor, feedback vizual. Acesta din urma este foarte important fiind un mijloc bun de informare.



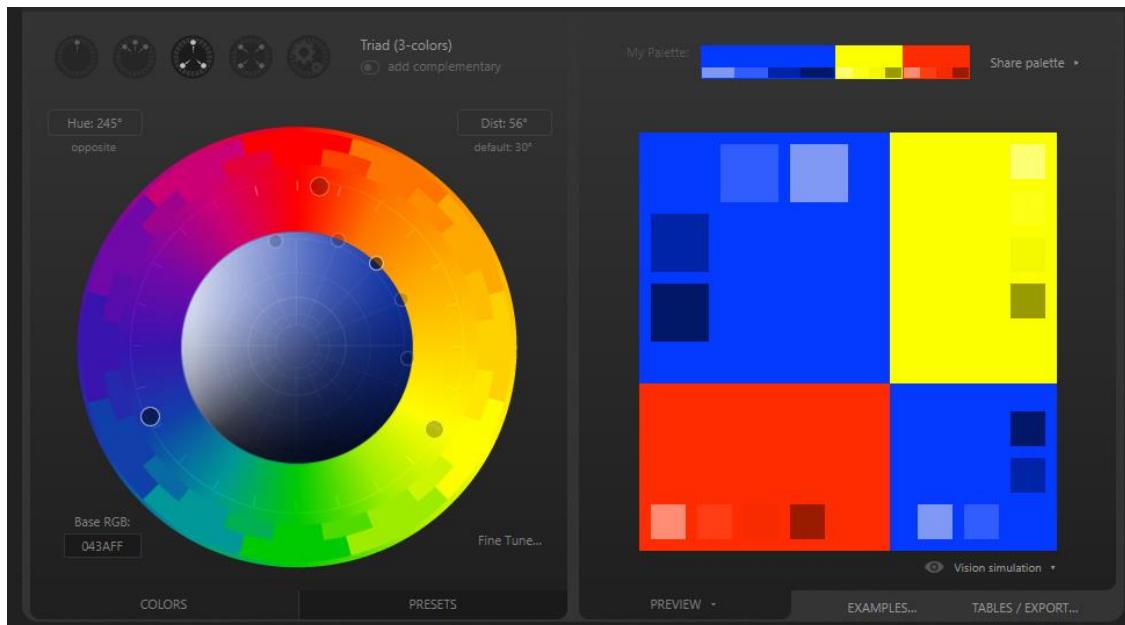
- **Designul trebuie sa fie simplu si intuitiv**, utilizatorii folosind de fiecare data intuitia. Steve Krug spunea ca principalul motiv pentru care utilizatorii folosesc intuitia atunci cand aceseaza o aplicatie este ca nu le pasa. *“If we find something that works, we stick to it. It doesn’t matter to us if we understand how things work, as long as we can use them”*. S-a constatat ca utilizatorii nu citesc informatia, ci doar o scaneaza, cautand cateva puncte de referinta care sa ii ghidizeze prin continutul paginii.
- **Spatiile albe sunt importante** – informatia este perceputa mult mai repede deoarece un utilizator, dupa cum am mentionat anterior, scaneaza informatia si o imparte in sectiuni usor de interpretat. Structurile complexe sunt greu de citit, scanat, analizat si lucrat cu ele.

## Alegerea culorilor potrivite

Alegera culorilor in dezvoltarea unei aplicatii web, de cele mai multe ori, nu este atat de simpla. In continuare o sa urmarim cele mai importante reguli in acest sens: (<https://flatuicolors.com/>)

- **Tehnica 60-30-10** – aceasta tehnica se aplica si in decorarea caselor si este una simpla. Pentru a fi in balanta, culorile trebuie combinate in proportie de 60%-30%-10%. Cea mai mare parte reprezinta culoarea dominanta, urmata de culoarea secundara, iar procentul de 10% il reprezinta culoarea care ajuta la realizarea accentelor in aplicatie
- **Contrastul** – prin folosirea acestuia se aduce individualitate fiecarui element de interfata, facandu-le pe fiecare in parte vizibile si accesibile. Contrastul este folosit pentru punerea in evidenta a anumitor sectiuni sau pentru butoane de tip call-to-action
- **Psihologia culorilor** – fiecare culoare influenteaza utilizatorul intr-un anumit fel si livreaza catre acesta un anumit mesaj. De exemplu:
  - **rosu** simbolizeaza atat un sentiment pozitiv (incredere) cat si un sentiment negativ (erori, folosire incorecta a unui element)
  - **verde** simbolizeaza succes, calmitate, natura (un task executat cu succes, folosirea corecta a aplicatiei, etc)
  - **alb** reprezinta puritate si claritate (ofera sentimentul de simplitate in cadrul unei aplicatii. Cum a spus Steve Jobs, inspirat de Leonardo da Vinci: “simplitatea este sofisticarea absoluta”)

Culorile se pot alege folosind tool-uri existente. De exemplu, <https://paletton.com/> este o astfel de aplicatie care asista utilizatorul sa genereze paleta de culori corecte plecand de la o culoare de baza. Plecand de la culoarea de baza **albastru** observam ca aplicatia ne alege culorile complementare **rosu** si **galben** si genereaza si paleta de culori aferenta acestor culori.



# Dezvoltarea Aplicatiilor Web utilizand ASP.NET Core MVC

## Laborator 11

---

### EXERCITII:

1. Pornind de la notiunile studiate in cadrul Cursului 11, sa se adauge in proiect functionalitatea de afisare paginata a articolelor (**VEZI Curs 11 – Sectiunea Afisarea paginata**).
2. Sa se integreze editorul de text Summernote (**VEZI Curs 11 – Sectiunea Includerea unui editor de text**).
3. Sa se implementeze motorul de cautare. Articolele pot fi cautate dupa titlu, continut sau dupa continutul comentariilor asociate acestora (**VEZI Curs 11 – Sectiunea Functionalitatea de cautare**).
4. **Fonturile** din platforma – pentru alegerea fonturilor se poate utiliza <https://fonts.googleapis.com/>

Se cauta sau se aleg fonturile dorite (ex: Roboto, Montserrat, Permanent Marker). Pentru adaugare se apasa acel “plus” din dreptul fontului. In meniu, in partea dreapta sus exista optiunea ***View selected families***. In aceasta fereastra, in sectiunea @import, se copiaza secventa de cod din interiorul tagului <style> (fara tagul <style>) si se adauga in proiect, in wwwroot → site.css → pe prima linie. Dupa acest pas, se pot utiliza aceste fonturi cu ajutorul atributului **font-family**.

```
font-family: 'Roboto', sans-serif;
font-family: 'Montserrat', sans-serif;
font-family: 'Permanent Marker', cursive;
```

5. **Stilizare logo** – pentru logo se poate utiliza fontul **Permanent Marker** si se poate stiliza cu ajutorul unei clase (in site.css) dupa care se foloseste clasa in \_Layout.cshtml.

De exemplu:

### Layout.cshtml

```
<a class="navbar-brand font-logo" asp-area="" asp-
controller="Articles" asp-action="Index">ArticlesApp</a>
```

### **wwwroot - site.css**

```
.font-logo {
    font-family: 'Permanent Marker', cursive;
}
.font-logo:hover {
    color: #00b894 !important;
}
```

La fel se include in site.css si celelalte doua fonturi (din cadrul exercitiului 4):

```
.font-montserrat {
    font-family: 'Montserrat', sans-serif;
}

.font-roboto {
    font-family: 'Roboto', sans-serif;
}
```

In footer se utilizeaza aceeasi clasa pentru afisarea logo-ului. Codul existent in footer este:

```
<footer class="border-top footer text-muted">
    <div class="container">
        © 2022 – ArticlesApp – <a asp-area="" asp-
controller="Home" asp-action="Privacy">Privacy</a>
    </div>
</footer>
```

Seventa de cod se modifica, folosind clasa font-logo pentru logo. De asemenea, se afiseaza si anul folosind metoda `DateTime.Now.Year`. Codul devine:

```
<footer class="border-top footer text-muted">
    <div class="container">
        <p>© @DateTime.Now.Year – <span class="font-
logo">ArticlesApp</span></p>
```

```
</div>
</footer>
```

6. **Footerul** – sa se stilizeze footer-ul, astfel incat sa ramana fix in partea de jos a paginii.

```
.footer-position {
    position: fixed;
    left: 0;
    bottom: 0;
    width: 100%;
    background-color: #ffffff;
    text-align: center;
    box-shadow: 0 3px 12px #636e72;
}
```

7. **Culorile folosite** – pentru alegerea culorilor se poate utiliza <https://flatuiccolors.com/>
8. **Pagina principală** – pe pagina principală /Home/Index să se afiseze pentru **utilizatorii neautentificati** doar 3 articole, în funcție de data publicării. Pentru a verifica dacă utilizatorul este autentificat se folosește în metoda Index condiția `User.Identity.IsAuthenticated`.

Dacă este autentificat se va face redirect către **metoda Index** din Controller-ul **Articles** utilizând `RedirectToAction("Index", "Article")`.

Dacă utilizatorul nu este autentificat se preia primul articol folosind metoda `.First()`, iar pentru a prelua următoarele două articole se folosește `Skip(1).Take(2)` deoarece nu trebuie să afișam încă o dată primul articol.

Primul articol să se afiseze folosind structura de mai jos. Să se utilizeze pentru titlu și continu fonturi diferite. De exemplu, pentru titlu fontul Roboto, iar pentru continut fontul Montserrat. Primul articol se va afisa pe un singur rand și pe o singura coloană, iar următoarele două articole pe un singur rand, dar pe două coloane. Pentru fiecare articol să se afiseze un link “Vezi articolul” .

De asemenea, pentru articolele de pe pagina principală să nu se afiseze tot continut. Să se afiseze doar câteva randuri din continutul fiecarui articol.

Apasarea logo-ului trebuie sa faca redirect catre HomeController → Index.

Ruta default trebuie sa fie configurata pentru a face redirect catre HomeController → Index.

9. **Meniul** (Views -> Shared -> \_Layout.cshtml) va afisa link-urile in functie de rolul pe care il are utilizatorul:

- **Utilizatorii neautentificati** nu o vada decat logo-ul aplicatiei (la apasarea acestuia, utilizatorii o sa fie redirectionati catre pagina Index din HomeController)
- **Utilizatorul cu rolul Admin** poate accesa “Afisare articole”, “Afisare categorii”, “Adaugare articol”, “Afisare utilizatori”, “Colectii”
- **Utilizatorii cu rolul Editor** pot accesa “Afisare articole”, “Adaugare articol”, “Colectii”
- **Utilizatorii cu rolul User** pot accesa “Afisare articole” si “Colectii”

### Implementare:

#### HomeController → metoda Index

```
public IActionResult Index()
{
    if (User.Identity.IsAuthenticated)
    {
        return RedirectToAction("Index", "Articles");
    }

    var articles = from article in db.Articles
                  select article;

    ViewBag.FirstArticle = articles.First();
    ViewBag.Articles = articles.OrderBy(o =>
o.Date).Skip(1).Take(2);

    return View();
}
```

## Views → Home → Index.cshtml

```

<div class="row">
    <div class="col-12">
        <div class="bg-light rounded-3 p-5 mb-4">
            <div class="row">
                <div class="col-md-8">
                    <h1 class="font-roboto display-5 fw-bold">@Html.Raw(ViewBag.FirstArticle.Title)</h1>
                    <div class="font-montserrat mb-3 truncate">@Html.Raw(ViewBag.FirstArticle.Content)</div>
                    <a href="/Articles/Show/@ViewBag.FirstArticle.Id" class="btn btn-outline-success">Vezi articolul</a>
                </div>
                <div class="col-md-4">
                    
                </div>
            </div>
        </div>
    </div>
</div>

<div class="row mb-4">
    @foreach (var article in ViewBag.Articles)
    {
        <div class="col-6 row-style">
            <div class="card">
                <div class="card-title p-3 text-center bg-dark text-white">
                    <h4>@Html.Raw(article.Title)</h4>
                </div>

                <div class="card-body">
                    <div class="font-montserrat truncate">@Html.Raw(article.Content)</div>
                </div>
                <div class="card-footer">
                    <a class="btn btn-outline-success" href="/Articles/Show/@article.Id">Vezi Articolul</a>
                </div>
            </div>
        </div>
    }
</div>

```