Dezvoltarea Aplicațiilor Web utilizând ASP.NET Core MVC
Curs 9

Cuprins

CEZARA BENEGUI

Sistemul de autentificare

Framework-ul ASP.NET Core oferă posibilitatea integrării unui sistem de autentificare, folosind **Identity Framework**. Acesta este un **sub-framework**, care funcționează în cadrul platformei **ASP.NET Core**. Este bine integrat cu alte părți ale acesteia, cum ar fi **Entity Framework Core**, pentru operațiuni legate de baze de date.

Identity Framework este compus dintr-o **suită de clase** și **secvențe de cod**, care facilitează implementarea rapidă a unui sistem de autentificare complex. Acest sistem oferă posibilitatea autentificării folosind user și parolă, alocarea de roluri pentru utilizatori, autentificarea folosind conturi third-party (autentificare prin rețele de socializare – Google, Facebook, Twitter etc.). De asemenea, Identity include și posibilitatea creării și manipulării rolurilor pe care le pot avea utilizatorii. **ASP.NET Core Identity** utilizează baza de date SQL Server pentru stocarea utilizatorilor, a rolurilor, dar și pentru asocierea dintre useri și roluri.

ASP.NET Core Identity

Pentru a genera un proiect care include componenta **Identity** pentru autentificare, trebuie selectată, la crearea proiectului, forma de autentificare: **Individual Accounts**. Această opțiune configurează proiectul, astfel încât să utilizeze **ASP.NET Core Identity**, un framework dedicat gestionării ***autentificării*** și ***autorizării***.

Individual Accounts permite autentificarea utilizatorilor prin crearea unor conturi proprii, stocate în baza de date asociată aplicației. Prin această metodă, fiecare utilizator are propriile credențiale (user și parolă), care sunt validate de aplicație.

CEZARA BENEGUI

Această configurare oferă beneficii precum:

- **Gestionarea securizată a parolelor** – Identity implementează hashing-ul și alte măsuri de securitate pentru protejarea parolelor;
- **Autentificare cu ajutorul serviciile externe** – se pot integra servicii third-party, cum ar fi Google, Facebook, Twitter, etc;
- **Sistem de roluri și permisiuni** – permite definirea și atribuirea de roluri utilizatorilor, ceea ce ajută la implementarea unui sistem de autorizare bazat pe roluri;
- **Interfață pentru managementul utilizatorilor** – se pot adăuga, edita sau șterge utilizatori și roluri cu ușurință;

Crearea unui proiect integrând Identity Framework

Windows:

- Se creează proiectul, utilizând template-ul ***ASP.NET Core Web App (Model-View-Controller).***
- Se adaugă sistemul de autentificare, selectând ***Individual Accounts*** (la fel cum am procedat în Cursul 6).

MAC și Linux:

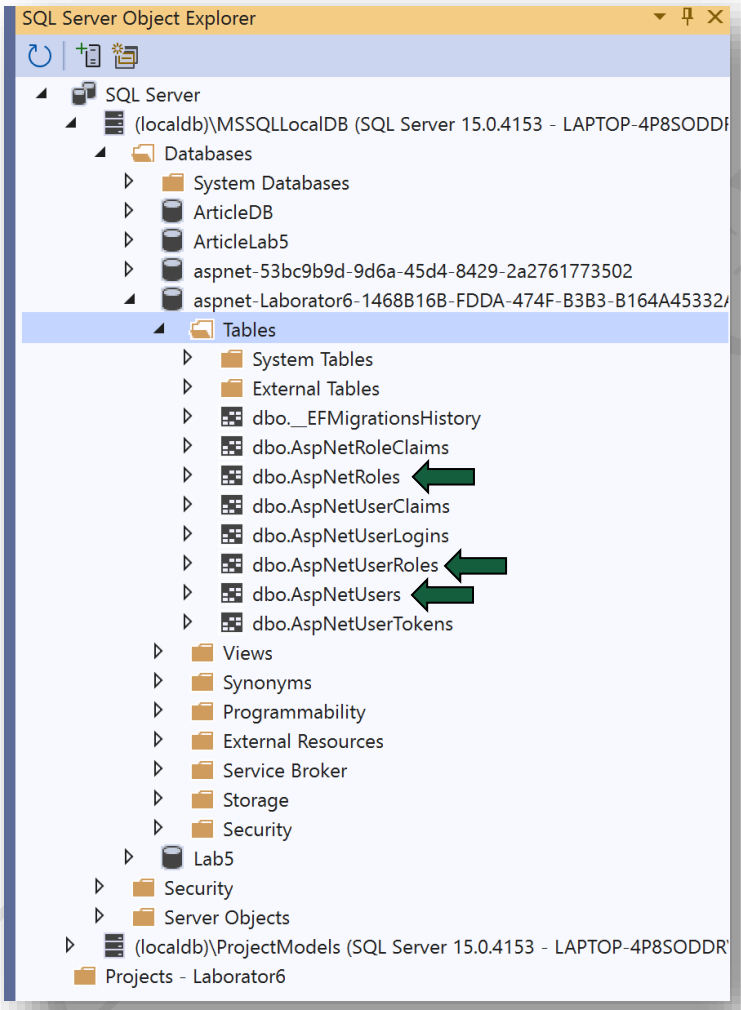
- Se utilizează comanda: `dotnet new webapp --auth Individual -o NumeProiect`
- Se rulează următoarele comenzi pentru sistemul de autentificare și entity framework:

- `dotnet tool install --global dotnet-ef --version 9.0.11`
- `dotnet add package Pomelo.EntityFrameworkCore.MySql`
- `dotnet add package Microsoft.EntityFrameworkCore`
- `dotnet add package Microsoft.EntityFrameworkCore.Design`
- `dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore`

CEZARA BENEGUI

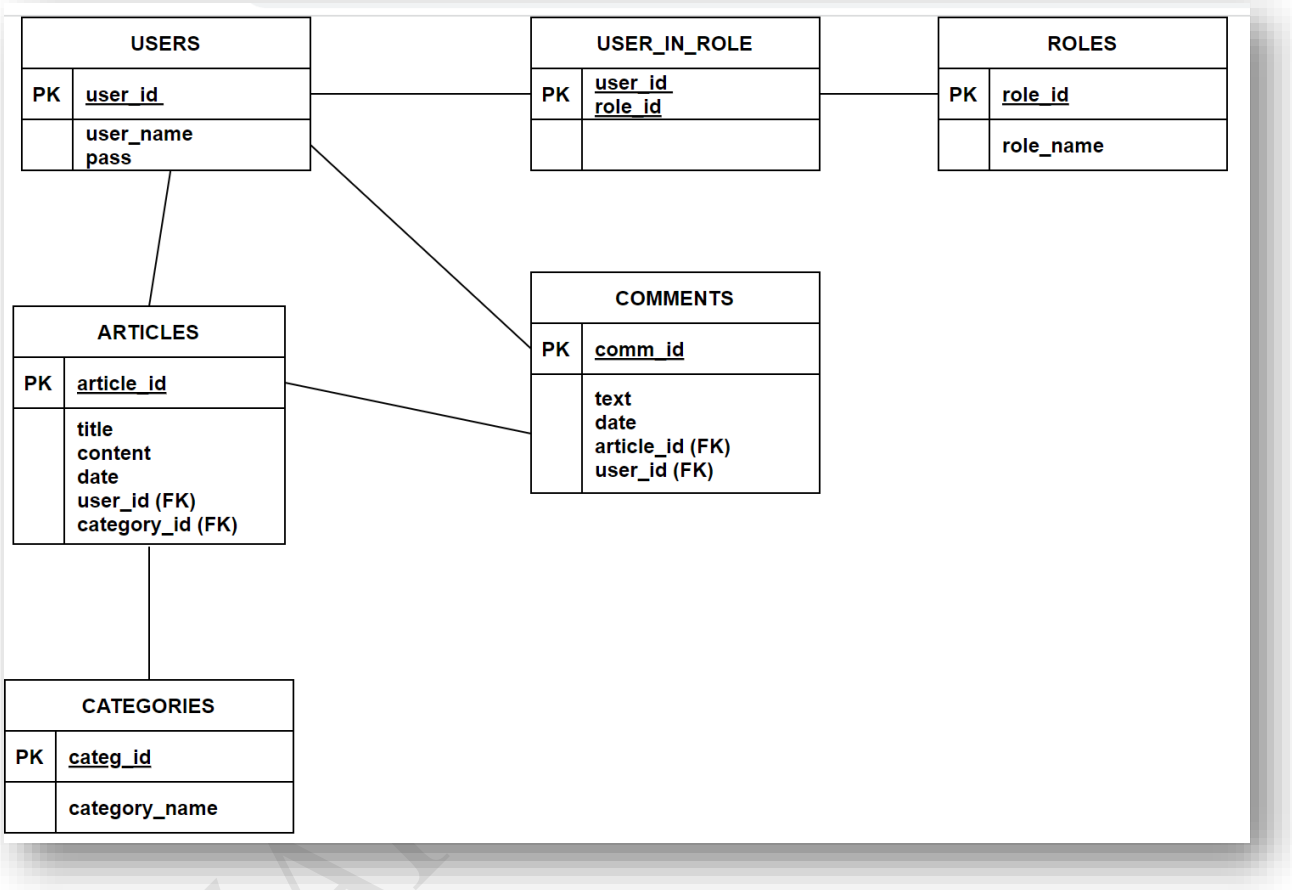
Adăugarea sistemului de autentificare a fost studiată în cadrul **Cursului 6 – Secțiunea Adăugarea sitemului de autentificare**.

După crearea proiectului împreună cu sistemul de autentificare, se pot vizualiza tabelele:



Roluri. Asocierea dintre roluri și utilizatori

Exemplele următoare sunt realizate în cadrul aplicației implementate în laborator (**Engine de știri**), conform următoarei diagrame:



Se consideră entitățile **Article**, **Category** și **Comment** cu următoarele proprietăți:

Article:

- **Id** – int → id-ul articolului (cheie primară);
- **Title** – string → titlul articolului este obligatoriu (Required), poate avea o lungime maximă de 100 de caractere (StringLength) și nu poate avea mai puțin de 5 caractere (MinLength);
- **Content** – string → conținutul articolului este obligatoriu (Required);

- **Date** – DateTime → data și ora la care este postat articolul;
- **CategoryId** – int → cheie externă – categoria din care face parte articolul este obligatorie (Required);
- **UserId** – string → cheie externă – reprezintă utilizatorul care a postat articolul;

Category:

- **Id** – int → id-ul categoriei (cheie primară);
- **CategoryName** – string → numele categoriei este obligatoriu (Required);

Comment:

Id – int → id-ul comentariului (cheie primară);

- **Content** – string → conținutul comentariului este obligatoriu (Required);
- **Date** – DateTime → data la care a fost postat comentariul;
- **ArticleId** – int (cheie externă) → articolul căruia îi aparține comentariul;
- **UserId** – string (cheie externă) → utilizatorul care a postat comentariul;

CEZARA BENEGUI

În continuare vom modifica anumite configurații existente și generate de ASP.NET Core Identity, astfel încât sistemul de autentificare să funcționeze și să putem configura rolurile din cadrul aplicației.

Pentru adăugarea rolurilor și pentru realizarea asocierii dintre utilizatori și roluri, trebuie parcurși următorii pași:

PASUL 1:

Pentru definirea utilizatorilor din aplicație, **Identity** include în baza de date un tabel **Users**, împreună cu toate atributele necesare (Id, UserName, Email, Password, PhoneNumber, etc).

În cadrul **Pasului 1**, se creează o nouă clasă în folderul **Models**, clasă care o să moștenească clasa de bază **IdentityUser** din pachetul ***Microsoft.AspNetCore.Identity***. În cazul în care dorim să extindem clasa **User**, adăugând atribute, putem realiza acest lucru în clasa pe care urmează să o implementăm.

Se adaugă o clasă pe care o numim **ApplicationUser**

```
namespace ArticlesApp.Models
{
      public class ApplicationUser : IdentityUser
      {

      }
}
```

Clasa pe care o moștenește, numită **IdentityUser**, este o clasă furnizată de framework-ul **ASP.NET Core Identity**. Aceasta definește modelul de bază pentru un utilizator dintr-un sistem de autentificare. Practic, reprezintă utilizatorul așa cum este el stocat în baza de date și oferă un set de proprietăți predefinite care sunt utilizate de ASP.NET Identity. Principalele proprietăți ale clasei IdentityUser sunt – Id, UserName, Emai, PasswordHash, etc.

CEZARA BENEGUI

Clasa **ApplicationUser** moștenește **IdentityUser**, ceea ce înseamnă că toate proprietățile și comportamentele predefinite ale lui **IdentityUser** sunt disponibile implicit în **ApplicationUser**.

Extinderea modelului de utilizator, prin crearea unei clase care moștenește **IdentityUser**, este utilă atunci când se dorește adăugarea unor atribute suplimentare (de exemplu: data nasterii, poza de profil, bio – o descriere a utilizatorului, etc).

PASUL 2:

În cadrul **Pasului 2** se adaugă serviciile necesare în fișierul **Program.cs**

```
builder.Services.AddDefaultIdentity<ApplicationUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
                .AddRoles<IdentityRole>()
                .AddEntityFrameworkStores<ApplicationDbContext>(
                );
```

În fișier există deja configurația pentru ***AddDefaultIdentity*** și ***AddEntityFrameworkStores***, fiind necesară doar adăugarea serviciului de management al rolurilor → ***AddRoles***.

Descrierea detaliată a serviciilor și metodelor:

- `AddDefaultIdentity<ApplicationUser>`

  - Serviciul pentru **User Interface**, **cookies** și toate mecanismele necesare pentru funcționarea autentificării;

  - Această metodă configurează o identitate implicită pentru utilizatori, bazată pe clasa personalizată ApplicationUser;

CEZARA BENEGUI

- `AddEntityFrameworkStores<ApplicationDbContext>`

  - Serviciul care realizează conexiunea cu baza de date, conectându-se la baza de date a cărui string de conexiune se afla în appsettings.json;

  - Această metodă configurează Identity pentru a folosi o bază de date gestionată de Entity Framework Core pentru stocarea informațiilor despre utilizatori și roluri;

  - **ApplicationDbContext** este clasa derivată din DbContext, care definește structura bazei de date și include setările pentru tabelele utilizatorilor, rolurilor, relațiilor dintre ele etc;

  - Cu această configurație, informațiile despre utilizatori, roluri, parole (hash-uite), token-uri și alte date specifice autentificării vor fi salvate în baza de date configurată în aplicație;

- `AddRoles<IdentityRole>`

  - Această metodă activează utilizarea de roluri în cadrul sistemului de autorizare. Rolurile sunt un mecanism prin care utilizatorilor li se pot atribui anumite permisiuni sau niveluri de acces, cum ar fi „Admin” sau „User”;

  - Prin intermediul acestei metode, se pot defini roluri și se pot atribui roluri utilizatorilor;

CEZARA BENEGUI

PASUL 3:

Contextul bazei de date se modifică astfel:

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options)
            : base(options)
        {
        }
    …
}
```

În folderul **Data** există contextul bazei de date, în clasa **ApplicationDbContext.cs**, care conține conexiunea cu baza de date și configurarea provider-ului de baze de date (în cazul nostru SQL Server), folosind dependency injection. **Contextul** este punctul central prin care se interacționează cu baza de date în Entity Framework Core.

În secvența următoare de cod, clasa **ApplicationDbContext** moștenește clasa de baza ***IdentityDbContext*** – clasă care se ocupă cu managementul userilor și rolurilor (conține proprietăți și metode cu ajutorul cărora se pot prelucra userii și rolurile din aplicație). ***IdentityDbContext*** este un context predefinit furnizat de ASP.NET Core Identity, care include toate tabelele și setările necesare pentru gestionarea autentificării și autorizării (utilizatori, roluri, relații între ele, etc.).

Clasa este de forma: `IdentityDbContext<TUser>`, ceea ce înseamnă că primește tipul clasei create pentru prelucrarea utilizatorilor din baza de date.

`(DbContextOptions<ApplicationDbContext> options)` – este un set de opțiuni care configurează conexiunea la baza de date. Acest obiect este injectat automat de ***Dependency Injection (DI)*** atunci când se aplică configurația bazei de date în ***Program.cs***.

`base(options)` – Apelează constructorul clasei de bază ***IdentityDbContext*** cu opțiunile furnizate. Aceasta asigură că toate configurațiile, cum ar fi conexiunea la baza de date, sunt transmise corect clasei de bază.

CEZARA BENEGUI

PASUL 4:

Se creează o nouă clasă, în folderul Models, cu ajutorul căreia se vor adăuga în baza de date ***rolurile*** necesare. În același timp, se pot asocia și utilizatori cu fiecare rol. Acest lucru se face o singură dată, urmând ca utilizatorii care creează cont să primească un rol în momentul înregistrării (logica la nivel de Controller).

Pentru crearea utilizatorilor inițiali, a rolurilor din aplicație, dar și pentru asocierea utilizatorilor inițiali cu rolurile, se creeaza o clasă, numită sugestiv **SeedData**, în cadrul căreia se va implementa o metodă numită **Initialize**. Prin intermediul acestei metode se vor popula cele trei tabele: ***Users***, ***Roles*** și ***UserRoles***.

Un ***Service Provider*** este utilizat pentru a injecta dependențele în aplicație (baza de date, sesiuni, pachete, autentificare, etc). În acest context, se utilizează serviciul responsabil de realizarea conexiunii la baza de date.

```
public static class SeedData
    {
        public static void Initialize(IServiceProvider
serviceProvider)
        {
            using (var context = new ApplicationDbContext(
                serviceProvider.GetRequiredService
                <DbContextOptions<ApplicationDbContext>>()))

            {
                // Verificam daca in baza de date exista cel putin un
rol
                // insemnand ca a fost rulat codul
                // De aceea facem return pentru a nu insera rolurile
inca o data
                // Acesta metoda trebuie sa se execute o singura data
                if (context.Roles.Any())
                {
                    return;   // baza de date contine deja roluri
                }
```

CEZARA BENEGUI

```
                // 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!")
                    },
```

CEZARA BENEGUI

```
                    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/

CEZARA BENEGUI

```
using (var context = new ApplicationDbContext(
                serviceProvider.GetRequiredService
                <DbContextOptions<ApplicationDbContext>>()))
```

- **Using**

  - Using este un bloc care garantează că obiectele create în interiorul său vor fi eliminate/distruse corect imediat ce nu mai sunt necesare;

  - În cazul contextului bazei de date (`ApplicationDbContext`), using este important pentru a închide conexiunile la baza de date și a elibera resursele alocate;

- **ApplicationDbContext**:

  - Este clasa contextului bazei de date;

  - Aceasta extinde funcționalitatea oferită de DbContext din Entity Framework Core și este utilă în gestionarea tabelelor, entităților și tranzacțiilor cu baza de date;

- **serviceProvider**

  - serviceProvider este un obiect care implementează interfața IServiceProvider;

  - Este un mecanism furnizat de ASP.NET Core pentru **Dependency Injection (DI)**;

  - Prin serviceProvider, se pot obține serviciile configurate în aplicație, inclusiv setările necesare pentru ApplicationDbContext;

CEZARA BENEGUI

- **GetRequiredService<DbContextOptions<ApplicationDbContext>>()**

  - Această metodă preia un serviciu specific din containerul de dependențe;

- **DbContextOptions<ApplicationDbContext>**

  - Este o configurație necesară pentru inițializarea ApplicationDbContext și include șirul de conexiune la baza de date, configurațiile specifice bazei de date (de exemplu, ***UseSqlServer*** pentru SQL Server), dar și alte setări pentru comportamentul contextului (de exemplu: Lazy Loading).

PASUL 5:

Prin instanța curentă a aplicației, se apelează, din clasa **SeedData**, metoda **Initialize**, ducând la crearea rolurilor și a utilizatorilor în baza de date.

În **Program.cs** se adaugă **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);
}
```

Această metodă creează un scop temporar de servicii (service scope) pentru a accesa și utiliza serviciile definite în containerul de Dependency Injection (DI). Această tehnică este utilizată în principal pentru a inițializa anumite resurse sau date în aplicație, în acest caz apelând metoda SeedData.Initialize(services).

CEZARA BENEGUI

- `app.Services.CreateScope()`

    - Creează un scop temporar în cadrul containerului de servicii;

    - În aplicațiile ASP.NET Core, serviciile înregistrate în containerul de DI au un lifetime definit (transient, scoped, singleton). Când sunt necesare servicii scoped (care există doar pe durata unei cereri), se creează manual un scop în afara ciclului de viață normal al cererilor HTTP;

- `scope.ServiceProvider`

    - Se obține un service provider asociat scopului creat, care permite accesul la serviciile înregistrate în containerul de DI;

    - Serviciile din aplicație pot fi obținute prin acest service provider;

- `SeedData.Initialize(services)`

    - Apelează metoda statică ***Initialize*** din clasa ***SeedData***, transmițând serviciile obținute din container;

    - Se utilizează pentru popularea bazei de date cu date inițiale (de exemplu: roluri, utilizatori sau alte entități necesare pentru funcționarea aplicației). Se pot crea structuri predefinite (tabele sau relații în baza de date);

CEZARA BENEGUI

PASUL 6:

Se adaugă proprietățile în clasele din Model, astfel:

În clasa Article

```
// PASUL 6 - useri si roluri

   public virtual ApplicationUser User { get; set; } → un
articol apartine unui singur utilizator
```

În clasa Comment

```
// PASUL 6 - useri si roluri

   public virtual ApplicationUser User { get; set; } → un
comentariu apartine unui singur utilizator
```

PASUL 7:

În folderul ***Views → Shared → _LoginPartial.cshtml →*** se modifică astfel:

```
@inject SignInManager<IdentityUser> SignInManager
@inject UserManager<IdentityUser> UserManager
```

```
@inject SignInManager<ApplicationUser> SignInManager
@inject UserManager<ApplicationUser> UserManager
```

CEZARA BENEGUI

PASUL 8:

Se realizează o nouă migrație în baza de date.

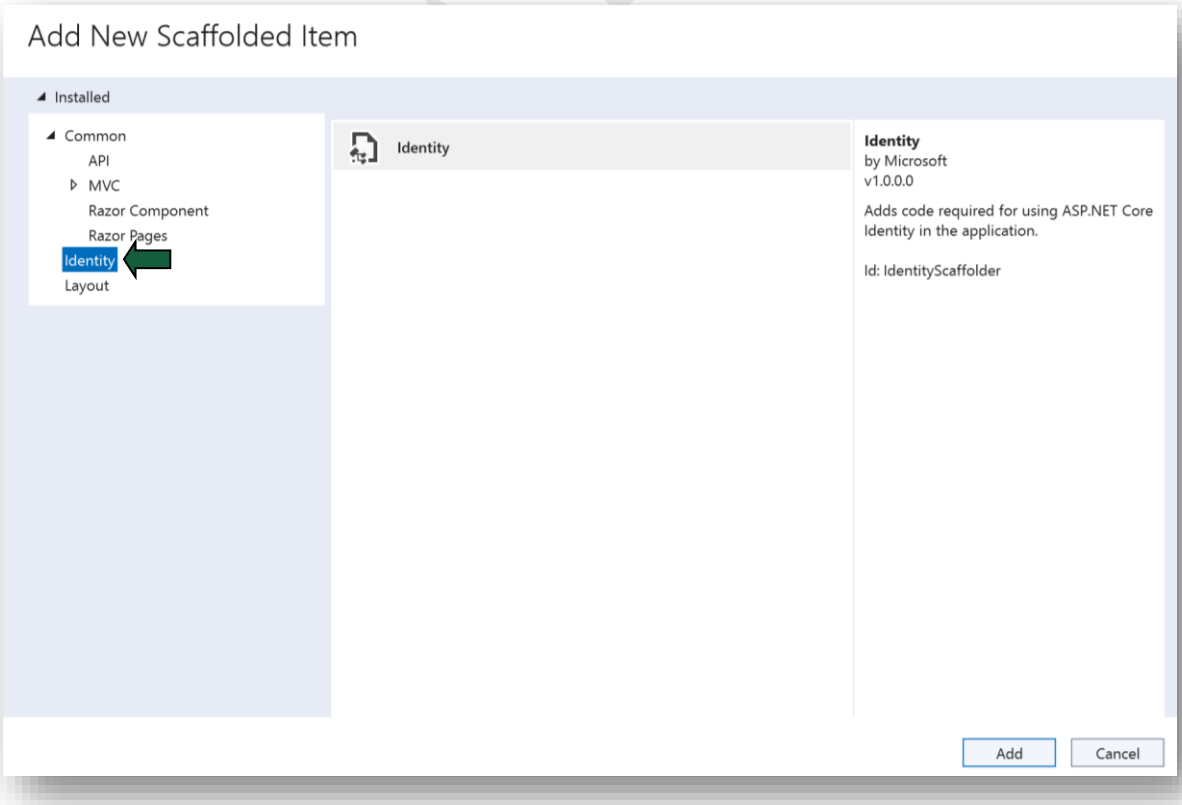
Se rulează pe rând:

1. Add-Migration DenumireMigratie
2. Update-Database

PASUL 9:

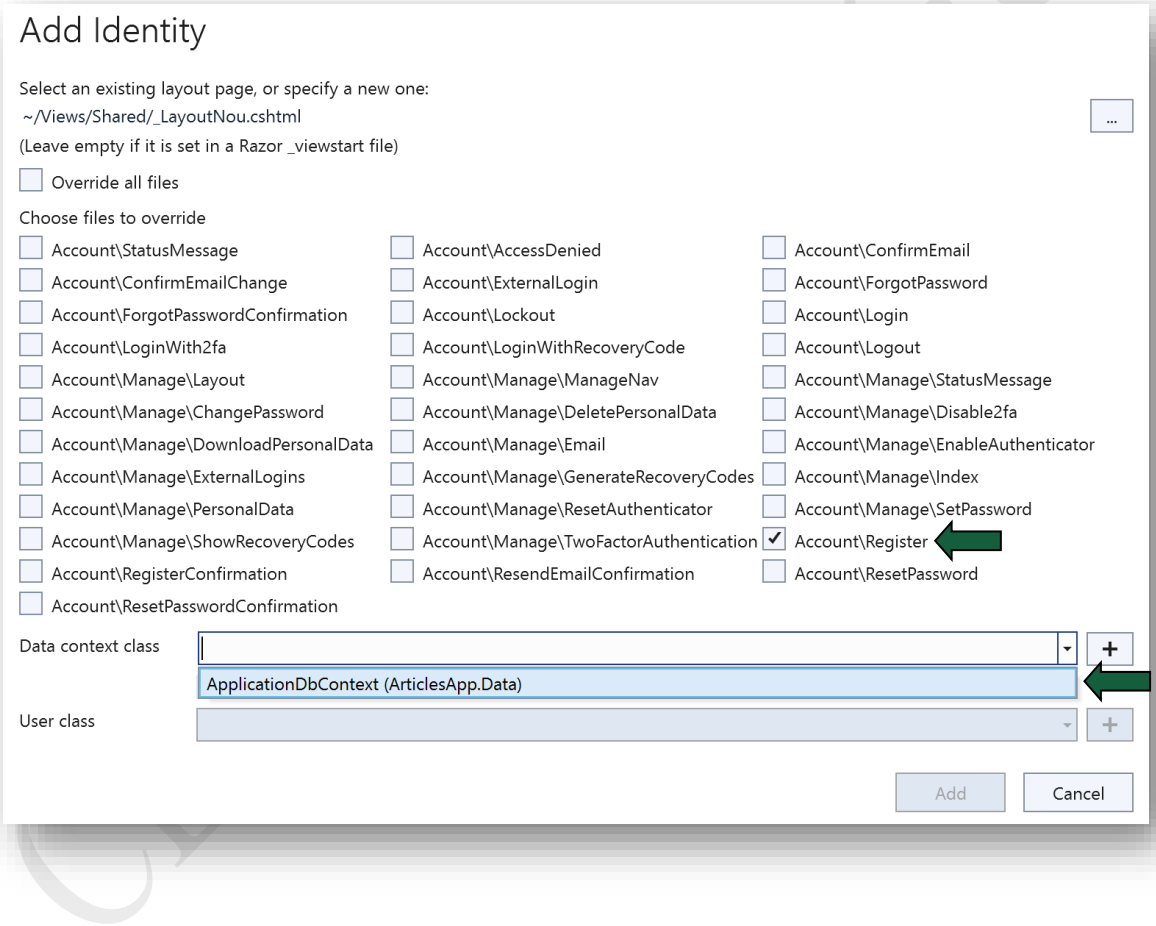
Pentru modificarea funcționalităților implementate în framework, se poate utiliza opțiunea **Add Scaffolded Item**, prin intermediul căreia se poate aduce în folderele de lucru codul sursă.

Click dreapta pe numele proiectului din Solution Explorer → Add → New Scaffolded Item



Pentru exemplul din cadrul cursului, avem nevoie de funcționalitatea de înregistrare. În momentul în care un utilizator se înregistrează în aplicație, acesta trebuie să primească un rol. În cazul aplicației ***Engine de știri***, utilizatorii înregistrați o să primească rolul **User**. Userii înregistrați pot vedea articolele, pot lăsa comentarii, pot edita și șterge propriile comentarii. Utilizatorii devin editori numai în momentul în care Adminul schimbă rolul.

Pentru adăugarea rolului de User în momentul înregistrării, trebuie modificată metoda **Register**.



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 și roluri. În Controller-ul **ArticlesController** se implementează următoarea secvență 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;
  }

…
 }
```

Documentație Microsoft – User Manager: https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.identity.usermanager-1?view=aspnetcore-8.0

Documentație Microsoft – Role Manager: https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.identity.rolemanager-1?view=aspnetcore-8.0

CEZARA BENEGUI

Atributul [Authorize]

Pentru a restricționa accesul utilizatorilor la metodele din cadrul unui Controller, se utilizează atributul [**Authorize]** la nivelul Controller-ului respectiv.

Pentru a restricționa accesul în cadrui fiecărei metode, se utilizează același 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 și Delete utilizând roluri

În cazul metodelor de adăugare, editare și ștergere, trebuie să se țină cont și de Id-ul utilizatorului care adaugă, editează sau șterge articolul. În cazul adăugării, o să se preia Id-ul utilizatorului pentru a fi adăugat în baza de date. În cazul editării, se verifică dacă utilizatorul care dorește să editeze articolul este utilizatorul căruia îi aparține articolul respectiv (adică utilizatorul care a postat articolul). Asemănător, se procedează și în cazul ștergerii unui articol. Un user poate șterge un articol doar dacă respectivul articol îi aparține. Un utilizator nu poate șterge articole postate în platformă de alți utilizatori.

CEZARA BENEGUI

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);
    }
}
```

În exemplul anterior, se poate observa cum, cu ajutorul managerului de useri, este preluat Id-ul userului curent, folosind metoda `GetUserId`. Astfel, în momentul inserării articolului în baza de date, se inserează și Id-ul utilizatorului care a postat articolul respectiv.

```
_userManager.GetUserId(User);
```

În cazul editării, se verifică dacă utilizatorul care dorește să modifice articolul este utilizatorul care a postat articolul. Preluarea Id-ului userului curent se realizează tot cu ajutorul metodei `GetUserId`. De asemenea, deoarece Administratorul are drepturi depline asupra aplicației, se verifică și rolul. Dacă utilizatorul are rolul ***Admin***, atunci el poate edita articolul. Pentru verificarea rolului, se utilizează metoda ***IsInRole("NumeRol")***.

CEZARA BENEGUI

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 procedează și în cazul editării cu HttpPost, dar și în cazul ștergerii. Se verifică dacă utilizatorul care intenționează să editeze sau să șteargă articolul este userul care a creat respectivul articol. În plus, se verifică și rolul utilizatorului care efectuează acțiunea. Dacă acesta are rolul ***Admin***, atunci poate efectua operațiuni C.R.U.D. asupra oricărei entități din aplicație.

CEZARA BENEGUI

Stocarea fișierelor în baza de date

Exemplul este realizat în cadrul proiectului dezvoltat în laborator. Modificările vor fi explicate punctual.

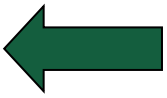
În cadrul modelului Article se adaugă o proprietate, numită sugestiv ***Image***, reprezentând calea imaginii.

```
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; }
}
```

În View-ul New, asociat Controller-ului ArticlesController, se va adăuga input-ul pentru încărcarea imaginii sau videoclipului. De asemenea, se modifică formularul, astfel încât encoding type-ul să primească valoarea `multipart/form-data`. Enctype – definește cum datele formularului sunt codificate înainte de a fi trimise către server. Este utilizat în mod special atunci când se dorește trimiterea de fișiere (cum ar fi imagini), împreună cu alte date de formular (obiecte, text, etc). Este necesar atunci când formularul include un câmp de tip `<input type="file" />`

CEZARA BENEGUI

```
<form enctype="multipart/form-data" asp-controller="Articles" asp-
action="New">

    <div class="form-group">
        <label asp-for="Title" class="form-label">Titlu Articol</label>
        <br />
        <input asp-for="Title" class="form-control" />
        <span asp-validation-for="Title" class="text-danger"></span>
        <br /><br />
    </div>

    <div class="form-group">
        <label asp-for="Content" class="form-label">Continut Articol</label>
        <br />
        <textarea asp-for="Content" class="form-control"></textarea>
        <span asp-validation-for="Content" class="text-danger"></span>
        <br /><br />
    </div>

    <div class="form-group">
        <label for="CategoryId">Selectati categoria</label>
        <br />
        <select asp-for="CategoryId" asp-items="Model.Categ" class="form-
control">
            <option value="">Selectati categoria</option>
        </select>
        <span asp-validation-for="CategoryId" class="text-danger"></span>
        <br /><br />

    </div>

    <div class="form-group">
        <label asp-for="Image">Incarca Imaginea</label>
        <input asp-for="Image" class="form-control" type="file" />
        <span asp-validation-for="Image" class="text-danger"></span>
    </div>

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

</form>
```



CEZARA BENEGUI

ArticlesController – metoda New cu [HttpPost]:

În continuare se va modifica metoda New, cea cu [HttpPost], din ArticlesController, astfel:

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

    private readonly IWebHostEnvironment _env;


// In constructor se face Dependency Injection

    public ArticlesController(ApplicationDbContext context,
IWebHostEnvironment env)
    {

          // Alocam conexiunea (injectata) cu baza de date unei
proprietati locale pentru a fi refolosita in metodele Controller-ului

        db = context;
        _env = env;
    }


// POST: Procesează datele trimise de utilizator

[HttpPost]
public async Task<IActionResult> New(Article article, IFormFile Image)
{
    article.Date = DateTime.Now;


    if (Image != null && Image.Length > 0)
    {
        // Verificăm extensia
        var allowedExtensions = new[] { ".jpg", ".jpeg", ".png", ".gif",
".mp4", ".mov" };

        var fileExtension = Path.GetExtension(Image.FileName).ToLower();

        if (!allowedExtensions.Contains(fileExtension))
        {
            ModelState.AddModelError("Image", "Fișierul trebuie să fie o
imagine (jpg, jpeg, png, gif) sau un video (mp4,  mov).");
            return View(article);
        }
```

Proprietatea din Model

CEZARA BENEGUI

Se creează un folder în wwwroot, numit images

```
        // Cale stocare
        var storagePath = Path.Combine(_env.WebRootPath, "images",
Image.FileName);
        var databaseFileName = "/images/" + Image.FileName;


        // Salvare fișier
        using (var fileStream = new FileStream(storagePath, FileMode.Create))
        {
            await Image.CopyToAsync(fileStream);
        }

        ModelState.Remove(nameof(article.Image));

        article.Image = databaseFileName;

    }

    if(TryValidateModel(article))
    {
        // Adăugare articol
        db.Articles.Add(article);
        await db.SaveChangesAsync();

        // Redirecționare după succes
        return RedirectToAction("Index", "Articles");
    }

    article.Categ = GetAllCategories();
    return View(article);
}
```

- **ModelState** este o colecție care reține starea validării pentru fiecare proprietate din model, inclusiv erorile asociate. Odată ce un câmp este procesat (de exemplu: **article.Image**), **ModelState** păstrează informațiile despre valoarea inițială și rezultatele validării.

- În exemplul anterior trebuie reactualizată starea deoarece câmpul article.Image este modificat, fiind setat la **databaseFileName**. Inițial, calea bazei de date are valoarea null.

- Astfel, dacă **ModelState** nu este actualizat (prin eliminarea intrării inițiale pentru article.Image), validarea ulterioară, cum ar fi ***TryValidateModel***, va folosi informațiile vechi, ceea ce poate duce la erori de validare.

CEZARA BENEGUI

- ***ModelState.Remove(nameof(article.Image))*** elimină starea asociată câmpului Image din ModelState, forțând sistemul să recalculeze validarea pentru acest câmp cu noua valoare (databaseFileName).

- După eliminarea intrării din ModelState, metoda TryValidateModel(article) revalidează întregul model, incluzând noua valoare a Image.

Pentru afișarea imaginii sau videoclipului în Index, se procedează astfel:

```
@foreach (var article in ViewBag.Articles)
{
    <div class="card">

        <div class="card-body">

            <h3 class="card-title alert-success py-3 px-3 rounded-
2">@article.Title</h3>

            @if(article.Image.Contains(".mp4") ||
article.Image.Contains(".mov"))
            {
                <video width="550" height="300" controls>
                    <source src="@article.Image" type="video/mp4">
                    Your browser does not support the video tag.
                </video>
            }
            else
            {
                <img src="@article.Image" alt="Article Image" style="width:
100%; height: 450px; object-fit: cover" />

            }

            <div class="card-text">@article.Content</div>

            <div class="d-flex justify-content-between flex-row mt-5">

                <div><i class="bi bi-globe"></i>
@article.Category.CategoryName</div>

                <a class="btn btn-success"
href="/Articles/Show/@article.Id">Afisare articol</a>

                <span class="alert-success px-1 align-content-
center">@article.Date</span>

            </div>
        </div>
    </div>
    <br />
    <br />
}
```